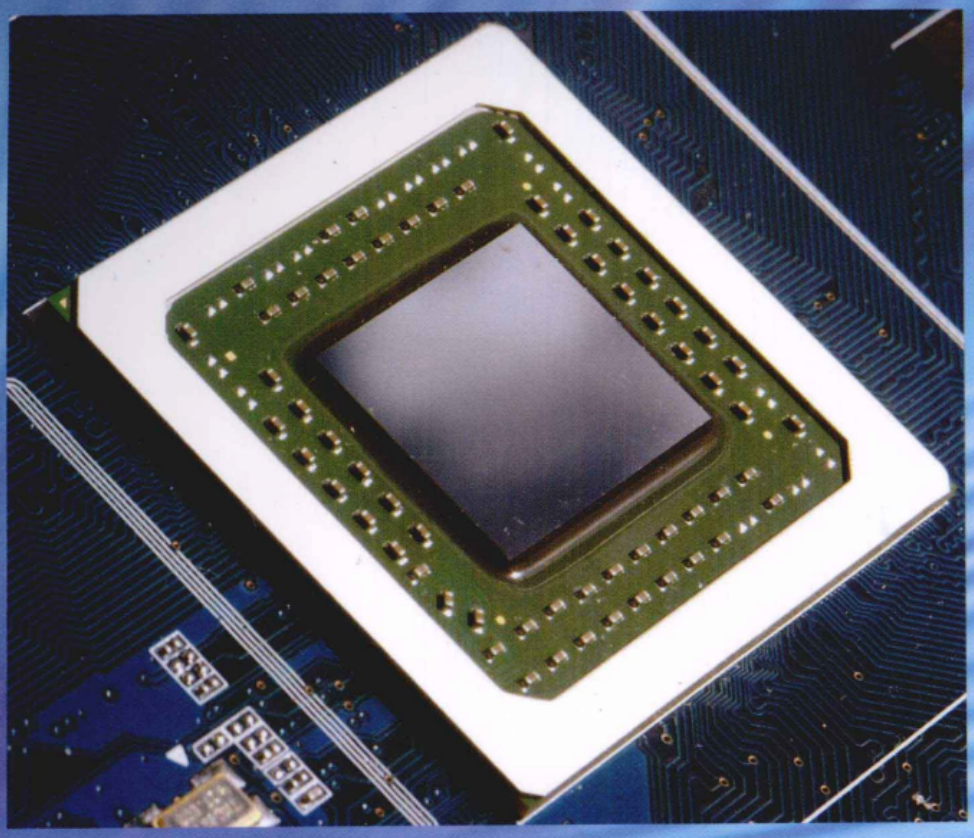
微电子与集成电路技术丛书

国家集成电路人才培养基地专家指导委员会组编



System-level FPGA Design and Application

系统级FPGA设计与应用

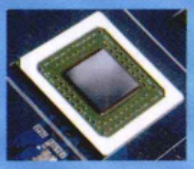
王伶俐　周学功　王　颖　编著

Wang Lingli　Zhou Xuegong　Wang Ying

童家榕　校

Tong Jiarong

清华大学出版社



System-level FPGA Design and Application
系统级FPGA设计与应用

本书特色

本书侧重于系统级FPGA的内核硬件结构，RTL级硬件编译的基本算法，基于FPGA的嵌入式操作系统和可重构系统设计基本的原理。具体内容包括：

第一，信息时代对并行计算的需求和系统级FPGA的并行阵列化硬件结构。从CPU、GPU、DSP和其他多核并行芯片的发展趋势理解FPGA内核的硬件结构，突出FPGA是细粒度的通用并行计算平台。

第二，基于FPGA的数字电路设计原理。介绍利用RTL级硬件描述语言的逻辑综合、工艺映射、布局布线和时序分析等基本算法。

第三，基于FPGA的嵌入式系统硬件和软件设计。介绍设备驱动的层次结构和自定义外设的开发，充分理解软硬件的底层接口技术。

最后，介绍基于FPGA的嵌入式操作系统及其可重构系统设计。理解嵌入式操作系统的基本特点和软硬件任务模型及其管理方式，讨论动态部分可重构系统的设计技术。

本书附录提供一些学生上机用的实验材料，有助于本科生和研究生的实践开发和动手能力的培养。

ISBN 978-7-302-27691-3
9 787302 276913 >
定价：29.00元

微电子与集成电路技术丛书
国家集成电路人才培养基地专家指导委员会组编

System-level FPGA Design and Application

系统级FPGA设计与应用

王伶俐　周学功　王　颖　编著
Wang Lingli　Zhou Xuegong　Wang Ying

童家榕　校
Tong Jiarong

清华大学出版社
北京

内 容 简 介

本书基于信息时代的特征和发展需求，分析并比较了各种可编程技术和可编程器件的特点，阐述了系统级 FPGA 的优越性，并介绍可编程逻辑器件的基础知识、基本原理和软硬件协同设计方法。本书并不从已有的商用 FPGA 器件和软件工具的角度介绍系统级 FPGA 的结构和应用技术，而是从可编程性这项核心技术出发，介绍了实现可编程性的底层硬件结构、设计数字电路所需要的 EDA 算法和软硬件协同设计技术，然后以商用 FPGA 器件和软件工具作为示例说明。这样可以把握商用器件结构及其开发环境的技术途径、发展趋势以及与其他信息技术的融合与交互过程。

全书共有 7 章。在第 1 章介绍数字信息技术平台后，第 2 章开始介绍与软件可编程性相对的各种硬件可编程技术和可编程硬件资源结构。第 3 章从通用型 CPU 的编译流程出发，介绍基于 FPGA 的数字电路设计流程和逻辑综合、工艺映射、布局布线、时序分析、基于 JTAG 的在线分析技术等内容。第 4 章和第 5 章分别介绍基于系统级 FPGA 的嵌入式系统的硬件和软件设计方法，主要讨论常见的微处理器、片上总线和自定义外设电路的设计方法和嵌入式系统软件开发技术。第 6 章介绍基于 FPGA 的可重构系统及其设计方法。第 7 章通过一个嵌入式系统设计实例对前面各章所学到的知识进行应用。本书附录部分还提供了一些上机材料。

本书适合高等院校或研究机构电子信息和计算机技术专业高年级大学生或研究生阅读，同样可供通信、机电类研究生、大学教师、电子电路设计和测试工程师等参考。

图书在版编目(CIP)数据

系统级 FPGA 设计与应用/王伶俐，周学功，王颖编著. --北京：清华大学出版社，2012.1
(微电子与集成电路技术丛书)
ISBN 978-7-302-27691-3

Ⅰ. ①系… Ⅱ. ①王… ②周… ③王… Ⅲ. ①可编程序逻辑器件—系统设计
Ⅳ. ①TP332.1

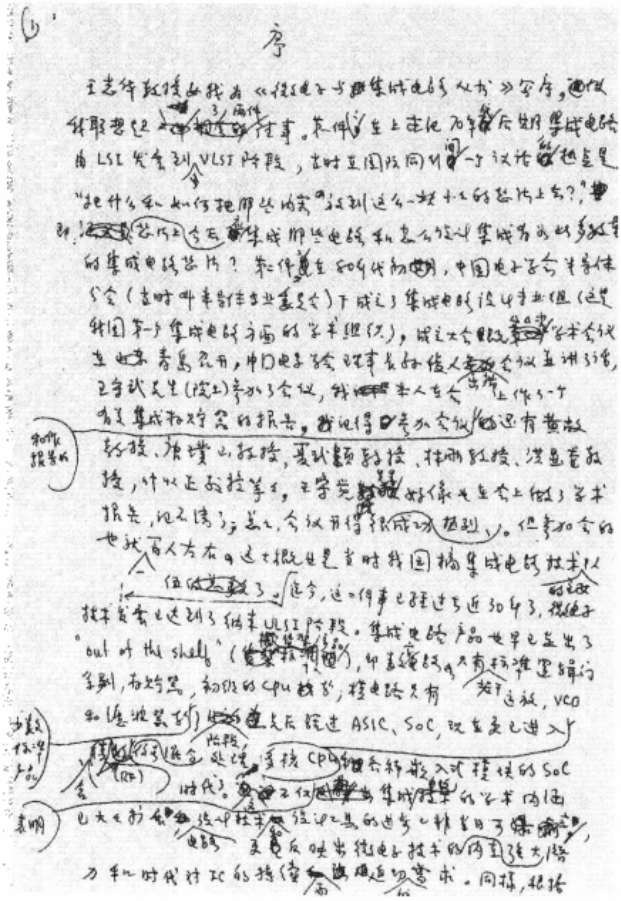
中国版本图书馆 CIP 数据核字(2011)第 267879 号

责任编辑：盛东亮
责任校对：焦丽丽
责任印制：杨　艳
出版发行：清华大学出版社　　**地　　址**：北京清华大学学研大厦 A 座
http://www.tup.com.cn　　**邮　　编**：100084
社　总　机：010-62770175　　**邮　购**：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质　量　反　馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
印 装 者：北京国马印刷厂
经　　销：全国新华书店
开　　本：185×260　**印　张**：14.75　**字　数**：364 千字
版　　次：2012 年 1 月第 1 版　　**印　次**：2012 年 1 月第 1 次印刷
印　　数：1～3000
定　　价：29.00 元

产品编号：042328-01

序一

王志华教授要我为《微电子与集成电路技术丛书》写序，使我联想起了两件往事。第一件：上世纪70年代后期，集成电路由LSI发展到VLSI阶段，当时在国际同行间一个讨论的热点是："把什么内容和如何把这些内容放到这么一块小小的芯片上去?"即今后芯片上应集成哪些电路和怎么设计集成有如此多电路的芯片？第二件：在上世纪80年代初，中国电子学会半导体分会（当时叫半导体专业委员会）下成立了集成电路设计专业组（这是我国第一个集成电路方面的学术组织），成立大会暨第一次学术会议在青岛召开，中国电子学会理事长孙俊人出席会议并讲了话，王守武先生（院士）参加了会议，我本人在会上作了一个有关集成存储器的报告，参加会议和作报告的还有黄敞教授、唐璞山教授、夏武颖教授、林雨教授、洪先龙教授、叶以正教授等等，记得王守觉先生也在会上作了学术报告。总之，会议开得很热烈、很成功。但参加会议的也就一百人左右，这大概也是当时我国搞集成电路技术的主要队伍。



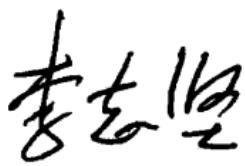
（李志坚院士为本丛书写的序言手稿）

迄今，这两件事已经过去近30年了，微电子技术已发展到了纳米ULSI阶段，集成电路产品也早已走出了out of the shelf的阶段，即数字电路只有若干标准逻辑门系列、存储器、初级的CPU等，模拟电路只有运放、VCO和滤波器等少数标准产品的阶段，先后经过ASIC、SOC，现今已进入了多核CPU、含射频、模拟与混合信号处理和各种嵌入式模块的SOC时代了。这不仅表明集成电路技术的学术内涵已大大扩展，电路设计技术和设计工具的进步已非当日可比，更反映出微电子技术的强大内在潜力和时代对IC的持续而迫切的需求。同样，根据中国半导体行业协会企业名册，我国有规模的IC设计企业已达到一百几十家，由此估计从业人员应该以万计算了，技术上我们已能独立设计出诸如3G手机核心芯片、嵌入式和高性能的CPU以及高档的保密芯

片等产品，这表明我国的集成电路设计产业和技术队伍也有了相应的很大进步。

微电子和集成电路是现代信息技术发展的基石，集成电路产业关系到国家的经济命脉、人民生活品质和国防与国家安全。作为现代主要高科技之一，集成电路技术方面的国际竞争十分激烈：谁的产品功能强、质量优、推出早、成本低，谁就占领主要市场，为胜者；谁落后一步，往往会被无情淘汰。夸大一些说，这一竞争往往是“只有第一，没有第二”。微电子和集成电路技术要求的基础知识十分广博，又与众多的高新技术相互交叉。集成电路产品更新换代极其迅速，产品从研制到投产周期日益缩短。这一切都决定了从业人员必须要有极高的业务素质，其中技术人员的基础知识、专业水平，特别是技术团队的创新能力更有决定性的作用。技术人员的知识基础，特别是新知识的补充，越来越重要；不仅在学校学习很重要，在工作中不断学习、不断充实更有必要。我想，国家集成电路人才培养基地专家委员会支持这套“微电子与集成电路技术丛书”的出版，除了要达到提高在校学生专业课程教学质量的目的外，更有这方面的深层意义。

丛书各分册的内容涵盖了微电子、数字和模拟集成电路的基本原理和技术知识，还包括了 RF 和数模混合信号处理、嵌入式和高性能处理器、低功耗芯片设计、SoC 设计方法学、EDA 工具及应用等广泛的现代专门课题内容。选题广阔、全面，符合与时俱进的精神。本丛书由清华大学王志华教授领衔的编审委员会组织编写，各册编写者主要是工作在第一线具有一定教学和实际工作经验的年轻学术骨干，同时聘请了一批国内同行中的资深专家为审稿人严格把关。我相信在这样老、中、青三代业内人士的共同努力下，本丛书的内容和质量是有保证的，它的出版一定会对我国集成电路人才培养和现有科技人员素质的提升起到促进作用。我更希望本丛书的编审一定要十分重视学术上的严谨性，并期盼，经过不断完善，至少有部分分册今后能成为教学的精品。



2010 年 1 月 10 日

序二

我曾经说过，每当我拿起笔为年轻学者出版一套丛书或一本书写序的时候，心中总是怀有特别的喜悦，因为这意味着辛勤耕耘后的丰硕收获，也意味着年轻的学者在进步与发展的道路上又迈出了新的一步，所以我总是乐意而为之。

自1958年TI公司的Jack S. Kilby和1959年仙童公司的Robert Noyce发明集成电路和硅平面集成电路以来，50年间，微电子和集成电路技术可谓发展神速，如同摩尔规律(Moore Law)所描述与预期的那样，按存储器算，集成度每18个月翻一番；就微处理器而言，集成度每两年翻一番；相应特征尺寸则缩小为上一技术节点的0.7。当前集成电路的集成度已从发明时的12个元件(2个晶体管、2个电容和8个电阻)发展到今天的数十亿个元件。集成电路功能日新月异，而成本迅速降低，微处理器上晶体管的价格每年平均下降约26%。2006年，Intel曾发表了一个很有意味的广告词："现在一个晶体管的价格大约与报纸上一个印刷字母的价格相当"。这就是说，人们只要买得起报纸，就消费得起集成电路。正因为如此，集成电路已广泛渗透到国民经济、国家安全建设和人民生活的各个领域，其应用的深度和广度远远超过了其他技术，是当代信息社会发展的基石。信息是人类社会三大资源之一，而且是目前利用得最不充分的资源。信息的本质是物质运动过程中的特征，信息技术包括信息的获取、传输、处理、存储、显示和随动执行等一系列的环节，而集成电路从狭义上讲则集信息处理、传输、存储等于一个小小的芯片中；从广义上讲，集成系统芯片(System on Chip，SoC)则集成了上述诸方面功能于一个芯片上或一个封装内的若干芯片(SiP)中，而这种可靠性高、功耗低的芯片又可以大批量、低成本地生产出来，因而势必大大地提高人们处理信息和应用信息的能力，大大地提高社会信息化的程度。它已如同细胞组成人体一样，成为现代工农业、国防装备和家庭耐用消费品不可分割的组成部分。集成电路科学技术的水平和它的产业规模也就理所当然地成为衡量一个国家或地区综合实力的重要标志之一，成为一个具有战略性的基础产业和高新科学技术领域。在过去的50年，在人类科学技术发展的沧海横流中，集成电路已经并正在不断显示其英雄本色。在人类社会步入信息化时代后，特别是在我国走"工业化带动信息化、信息化促进工业化"的具有中国特色的新型工业化道路中，在市场需求和国家中长期科学规划重大专项投入的双重促进下，我国集成电路科技和产业必将得到更多的发展机遇，带来更多的创新。

现代社会的科技竞争，包括微电子与集成电路技术的竞争，归根到底是人才的竞争。得人才者得天下，集人心者集大成，希望在人才。培育人才最重要的工作在于教育，只要人类社会存在，教育就是永恒的主题；只要人的生命存在，学习就是不竭的任务。不管是学校教育还是在实践基础上的自学进修都需要教材或称之为教本，所谓"教本、教本，乃教学之本"。

集成电路不是直接与消费者见面的最终产品，因而系统应用是使集成电路产生巨大增值的关键环节，而设计是微电子技术和集成电路产业链中最接近应用、也就是最接近市场的

领域，具有巨大的创新与市场空间。50 年来集成电路的发展史是需求牵引和科学发现、技术发明推动相结合的历史，是一部技术创新和机制创新的历史。需求牵引往往由市场和系统应用提出，而设计首先就需要面对这种新的需求。一个好的算法、标准和设计往往可以引领市场的发展，为微电子和集成电路开拓一个崭新的领域。因此，“微电子与集成电路技术丛书”首批启动就将重点放在与设计相关的专业课程是十分恰当的。

《微电子与集成电路技术丛书》由国家集成电路人才培养基地专家委员会主持编写，第一批启动 16 册，第二批将再启动 10 余册，其内容涵盖了微电子及集成电路领域的主要范畴，尤以设计为主体。由年轻的学科带头人、清华大学王志华教授领衔丛书编审委员会，参加编写的有 30 多位年轻的学科带头人和学术骨干，这反映了我国年轻一代学者正在茁壮成长。同时，丛书还邀请了一批治学严谨的年长一代科学家和学者担任审稿工作，在这些学者的名单中我看到了在上世纪 80 年代就曾共事过的如洪先龙教授、吉利久教授、张建人教授等老朋友。我坚信：由年轻学者执笔，由年长一代科学家把关，丛书学术内容的新颖性和严谨性就一定能得到可靠的保证。

这套丛书特别适合于微电子与集成电路专业高年级本科生、研究生阅读，也适合相关领域的工程技术人员作为参考书。我相信，阅读本丛书的学生和科技人员必将受益匪浅。

2010 年 1 月 5 日于北京大学

序三

有一个古老的中国寓言，说的是一个年轻的读书人看到一位仙翁用手指点一下石头，石头就能变成金砖，这是成语“点石成金”的来源。多年后的今天，人们常常只关注到那腐朽化神奇的“一点”而忘了故事寓意中最重要的一环，即练得此法术的方法和为此所需要付出的数十年的功德和修为。自 1995 年我从业以来，就一直惊叹微电子及集成电路是一个多么像“点石成金”的行业，而同时又是一个多么讲究方法、多么需要付出艰苦努力的领域！

多年来我和我在 Synopsys 公司的同事们一起在国内推广基于逻辑综合的自顶向下的集成电路设计方法，经历了逆向设计解剖版图的初始阶段，那时全国设计业产值不过上亿元人民币、设计企业不过数十家、从业人员以百十计，而现在，中国大陆已是全球最大的集成电路市场、全国设计业产值超过 300 亿（依然是方兴未艾）、设计企业超过 500 家、从业人员数以万计；从那时开设集成电路设计课程并装备集成电路设计工具环境的寥寥几所高校，到目前 19 所院校建有集成电路工程特色专业、20 个（含在建）集成电路人才培养基地、约 40 个大学招收集成电路工程硕士、近 50 个大学（所、系）配置了我公司的 IC 设计工具的大学计划包。这真是个天翻地覆的变化。IC 设计是个智力密集型、创新密集型的行业。没有高素质、实践型的人才和人才培养支撑体系，就没有持续发展的可能。人才依然是我们发展过程中遇到的最大瓶颈之一，我们仍然感到缺少一套系统化的、覆盖该领域最新技术的微电子及集成电路教材。公司总部有一个教材指导委员会（Curriculum Advisory Board），他们基于多年的研究积累，针对本科生和研究生主持开发了一套微电子及集成电路课程体系，当我了解到相应的教学课程内容后，便立即想到如果以此为参考帮助国内开发一套微电子及集成电路领域的教材和参考书，应该是非常有意义的。此想法得到了时任国家集成电路人才培养基地专家委员会主任委员浙江大学严晓浪教授和委员会副主任委员清华大学王志华教授的赞同，也得到了 Synopsys 公司全球总裁陈志宽博士的积极支持。一年多后的今天，我们终于见到了这套丛书第一批 16 本的面世！这是主编王志华教授和 30 多位编审者们辛勤劳动的成果，也要感谢李志坚院士、王阳元院士这样德高望重的多位业界前辈对丛书编著选题的把握、对方向的关注、对内容的裁夺等。我也非常高兴我的同事和我的公司在这件事情上面所作的微薄贡献。

一直以来，参与并推动中国集成电路产业的腾飞是我们的梦想。回望过去，中国每一天都在进步，中国集成电路产业每一年都在成长。世界范围内产业的大迁移、国内市场需求的强劲拉动、有利的产业政策和创业环境，正带给中国集成电路产业发展最佳的契机。而人才

培养是最重要的环节和基础，是漫长的付出和努力、是艰辛的孕育和耕耘、是由量变到质变的积累，直到腾飞前的化蛹成蝶。在老中青几代人的共同努力下，相信在不久的将来我们的行业一定会创造出一座座的金山、一定会拥有一大批“点石成金”手！“长风破浪会有时，直挂云帆济沧海”。我由衷地希望这套丛书的出版可以帮助实现我们共同的心愿，并殷切期待丛书下一批十多本著作的尽早面世！

2009 年 12 月于北京

主编序言

潘建岳先生和我是清华校友，一直以来，他和他的同仁对国内集成电路行业的发展给予了极大的关注和支持。2007 年初，时任 Synopsys 中国区总裁的潘建岳先生提出，将 Synopsys 公司教材指导委员会(Curriculum Advisory Board)主持开发的课程体系和一套以 IC 设计为主的教学课件赠送给国家集成电路人才培养基地专家委员会，期望对国内集成电路设计人才培养特别是教材建设有所帮助。当时，教育部和科技部已经批准在 20 所大学建立(含筹建)集成电路人才培养基地，国务院学位办已经批准在约 40 个大学招收集成电路工程领域的工程类硕士研究生，教育部也于 2007 年已经批准在 19 所院校建设微电子学专业集成电路领域的特色本科专业建设。除此之外，电子科学与技术、信息与通信工程、计算机科学与技术等学科的高层次人才，也都需要具备集成电路知识。受潘建岳先生的建议及赠送的材料的启发，集成电路人才培养基地专家委员决定编写《微电子与集成电路技术丛书》并委托我担任主编。

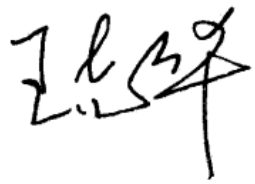
为做好丛书的编写工作，潘建岳先生和我一起专门拜访了王阳元院士，请求指导和支持。王阳元院士是我国杰出的教育家和科学家，为我国微电子事业的创立和发展做出了不可磨灭的功绩。得知我们计划编写一套《微电子与集成电路技术丛书》之后，王院士除了表示支持之外，还特别叮嘱我们关心图书的内容和质量。丛书要为读者提供完整的知识体系，提供正确和准确的技术内容，对于飞速发展和变化的微电子和集成电路领域，要力求反映最新的技术进展。但图书的价值，不仅体现在当前最新知识的传播上，在图书的技术内容过时之后，书籍依然承载着历史和文化的价值。

担任主编工作后我一直有一种忐忑不安的心情，主要是感到自己不足以把握日新月异的集成电路知识，更没有勇气面对王阳元院士讲的书籍的历史文化价值的承载作用。作为国家集成电路人才培养基地专家委员会中的一员，在诸多年高德劭的前辈的指派下，我诚惶诚恐地承担了这个任务。

我们邀请了国内在微电子和集成电路领域第一线工作的年轻学术骨干参加丛书编写。他们不但具有相当丰富的教学经验，而且活跃在相关科学研究的前沿，其中还有部分教师参加过国家集成电路人才培养基地专家委员会和国家外国专家局支持的技术培训。他们的知识、经验和奉献精神，是本丛书面世的基础；我们同时聘请了一批国内同行中的资深专家参加丛书编委会，他们除了为图书选题、内容取舍出谋划策之外，还作为审稿人对图书的技术内容、讲述方法甚至语言文字严格把关。他们的工作，不仅保证了图书编写质量，而且是对国内微电子和集成电路领域年轻才俊的大力扶持和帮助。感谢这些知识渊博、德高望重的前辈。感谢教育部高等教育司、科技部高新技术及产业化司、原信息产业部电子产品司的领导对图书编写和出版的支持，他们对教育、科技发展以及微电子行业需求的深入了解，使丛书的编写得以适应行业的需求。感谢浙江大学严晓浪教授，他作为国家集成电路人才培养

基地专家委员会的主任委员，始终关心和指导着丛书编写的各个环节。

现在，《微电子与集成电路技术丛书》第一批16种图书终于面世了！本丛书内容涵盖了微电子、数字和模拟集成电路的基本原理和技术知识，还包括了射频电路设计、数模混合信号处理、嵌入式和高性能处理器、低功耗芯片设计、SoC设计方法学、EDA工具及应用等广泛的现代专门课题内容。我们期望丛书不辜负微电子和集成电路领域专家的期望，以全面的选题、丰富的内容、准确的知识、科学的表述传播微电子和集成电路领域的知识，满足我国集成电路领域人才培养的需求。如果该丛书能为我国微电子和集成电路领域的科技发展作出点滴贡献，功劳属于图书的编写者以及为图书的面世贡献了力量的众多无名英雄。



2009年11月于北京清华园

前言

2004 年 Intel 公司由于功耗等技术原因取消了单核处理器的后续研发计划，标志着通用型 CPU 进入了多核时代，这就意味着 CPU 性能的提高必须突破之前主要依赖于半导体工艺技术进步的技术途径。在通用型处理器结构方面，ALU 算术逻辑单元所占的芯片面积不到 10%，这就是说 CPU 内部大部分的晶体管面积和功耗都不是直接用于算术逻辑运算，而是被多级高速缓存、指令译码、存储控制与管理、分支预测等辅助单元所占用，因此，通用型 CPU 的计算和功耗效率是非常低的。相比较而言，GPU 处理器面向图形处理等应用，大大简化了内部的高速缓存和控制逻辑单元，从而让更多的晶体管直接参与计算，使得一个芯片内部包含上百个计算核。在 2010 年 11 月公布的全球最快超级计算机 TOP500 榜单中，“天河一号”超级计算机采用了 GPU 加速结构而首次让中国的机器位于排行榜榜首。

但不管是通用型 CPU、GPU 或者 DSP 处理器，其核心结构都是基于 1945 年冯·诺依曼等人提出的“存储程序计算机”结构，其中存储单元和计算单元在硬件上就是分开的。处理器的运行过程包括取指令、译码、取数、执行、结果返回等流水线过程，其中只有中间的“执行”这一步骤是真正用于计算的。因此，为了充分发挥半导体工艺技术进步带来的成果，提高芯片的性能和能耗效率，必须从底层的核心硬件结构和执行方式进行突破，从而让大部分晶体管能够直接参与计算任务。

可编程逻辑器件从 20 世纪 80 年代以来发展非常迅速。由于早期可编程逻辑器件的接口功能强大，因此它主要用于胶合逻辑应用。后来逐步放弃了基于可编程与或阵列的核心结构，采用了基于查找表的可编程单元结构，从而让它的逻辑功能不断强大，成为专用集成电路原型验证的最佳途径。在 FPGA 结构中，查找表既可以作存储单元，也可以作为可编程逻辑单元。逻辑单元之间不再以总线的结构，而是以复杂的分布式层次化互连资源进行通信。这些结构上的特点就已经突破了“存储程序计算机”的局限性。至今可编程逻辑器件内部直接嵌入了高速通信 IO 核、微处理器核、存储器核、DSP 模块和复杂的时钟管理硬件模块，从而大大提高了芯片的数据通信与计算能力。我们把这种包含了基于细粒度查找表结构的可编程逻辑单元和各种粗粒度的数据通信与计算单元的现场可编程逻辑器件称为系统级 FPGA。为了充分发挥半导体工艺的优势，系统级 FPGA 采用了业界最领先的半导体工艺技术。从应用领域来看，系统级 FPGA 正在不断侵占 ASIC/ASSP、CPU、DSP 器件的市场，涵盖了通信、汽车电子、医疗设备、消费类电子、高性能计算、嵌入式系统、航空航天、工业控制等领域。

虽然基于配置流与硬件可编程技术的系统级 FPGA 具有很好的并行性和计算效率，在结构方面克服了“存储程序计算机”的结构劣势，在运行过程方面不再需要指令、译码、取数等指令流操作，但是它的硬件结构还不够成熟，工具链过于复杂。在传统的通用型 CPU 领域，GCC 开源编译器基本上支持各种商用处理器，能够产生二进制可执行文件。但是至今

为止还没有一个统一的 FPGA 编译器，能够支持各种系统级 FPGA 的可编程结构，从而产生正确的可编程配置文件。也没有一个成熟的工具链，能够有效地支持各种系统级 FPGA 的软硬件协同仿真、高层次综合和软硬件混合在线调试等技术。为了充分发挥系统级 FPGA 所提供的硬件优势及其工具链的作用，用户需要了解它的基本硬件结构、EDA 软件算法和软硬件协同设计等知识。本书结合复旦大学专用集成电路与系统国家重点实验室在可编程逻辑器件和可重构计算领域的项目开发经验和教学体会，分析比较了各种可编程逻辑器件的结构，定性地说明了基于查找表 FPGA 结构的优越性。在此基础了介绍了 FPGA 工具链后端 EDA 流程和基本算法，并且结合软硬件协同设计技术，提供了基于系统级 FPGA 的可重构系统和嵌入式系统设计实例。

在本书的写作和校对过程中，我们得到了国家重点实验室 CAD 教研室的老师和学生的多方帮助，他们有唐璞山教授、曾璇教授、赵文庆教授、来金梅教授、曹伟博士、谭道珍老师和研究生陈志辉、刘智斌、叶晓敏、邵海波、陈帅、陈佳临、王侃文、陈利光、包杰等。同时也向我们的家人表示歉意，我们没有在周末和假期的时候，安排足够的时间和家人一起度过，再次向他们的付出表示深切的感谢。

王伶俐　周学功　王颖　童家榕

2011 年 8 月

复旦大学张江校区

目录

图索引

表索引

第1章

数字信息技术平台

1.1 数字信息时代的发展需求

1995年美国麻省理工学院教授Negroponte(尼葛洛庞帝)在他的《数字化生存》一书中，讲述了一次饶有趣味的经历[1]。一天，他去参观美国一家集成电路制造公司，前台接待员询问他随身带的笔记本电脑的机型、序号和价值准备登记，他的回答是："大约值100万到200万美元吧！"这让接待员丈二和尚摸不着头脑，根据这位接待员的估计，这种型号的新电脑当时价值大约为2000美元。

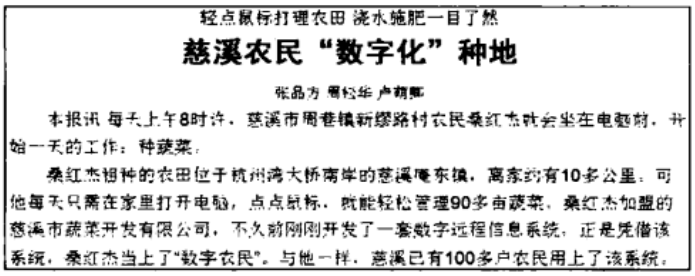
由此可见，当我们衡量一台电脑的价值时，其硬件部分的价值只占很少的一部分。计算机的主要价值体现在它内部所包含的信息，例如各种软件、文档、多媒体材料、联系方式和所有的电子邮件(E-mail)等信息。人类社会经过了农业、工业时代，如今已经进入了信息时代，每个人的日常生活方式都发生了巨大变化。根据EETimes(Electronics Engineering Times，电子工程专辑)在2008年3月的报导，美国Pew Internet Project进行抽样调查显示[2]：51%的美国人认为手机是日常生活中最不可缺少的，其次是互联网(占45%)、电视机(占43%)和有线电话(占40%)；而在此两年前调查结果的次序是：有线电话、电视机、手机和互联网。我们相信手机和互联网将进一步改变我们的日常生活方式。今天如果我们丢失了一部手机或一台笔记本电脑，那么让我们心痛的是它所存储的所有联系方式、多媒体材料和各种短信文档等——即它所保存的各种信息，这是一台新手机或新电脑所无法弥补的。

Negroponte在《数字化生存》一书中强调："计算不再只和计算机相关，它决定我们的生存。"这就是说，数字信息时代快速便利的信息计算和传播技术极大地改变了我们的生存方式。如图1-1所示，2008年4月1日的《新民晚报》报导，浙江慈溪的"数字农民"利用数字远程信息系统，每天坐在电脑前对离家10多公里的90多亩农田进行"数字化"管理。"数字农民"利用鼠标操控电子探头，可以在屏幕上看到农田全景，并检查菜叶上是否有虫子。实地的农田操作通过电话让帮工执行就可。由此可见，信息技术的发展对全球化的生活方式、价值体系、管理模式等产生了革命性的影响，这种影响在《世界是平的》一书中阐述得十分详尽[3]。

1.1.1 信息时代的来临及其特征

在人类生存和文明发展史上，文字与信息存储及其传播技术一直起着至关重要的作用。虽然信息时代需要文字、印刷术和基本的通信技术，但是它们的出现并不标志着信息时代的开始。在文字出现很久以后，随着活字印刷术的应用推广，欧洲的文艺复兴(Renaissance)、

宗教改革(Reformation)和思想启蒙(Enlightenment)运动使得文字信息传播在中世纪后得到空前的发展[4],但是历史还没有称之为"信息时代"。一直到第二次世界大战结束和冷战以后,随着科学技术的急速发展,特别是半导体集成电路、计算机、软件和网络通信技术的普及与更新换代,信息的获取、存储、变换处理和传输得到空前发展,信息产业已经和一个国家的综合国力、人民的日常生活密切相关,于是逐渐形成了一个崭新的时代,通常称做信息时代、后工业时代或第三次浪潮。信息时代在不知不觉中改变了我们的思维方式。例如,随着移动通信及互联网的不断普及,人们遇到一个问题时首先会想到在网上利用搜索引擎查找可能有的现成答案,而不是首先利用自己的大脑进行深入的独立思考。这种改变从一方面来说是一件好事,它可以很方便地共享信息。另一方面,这可能会逐渐地影响我们独立思考、刻苦钻研的良好习惯,人们会轻易接受网上传播的可能错误的信息。



轻点鼠标打理农田 浇水施肥一目了然

慈溪农民"数字化"种地

张品方 曹松华 卢朝辉

本报讯 每天上午8时许,慈溪市周巷镇新塘路村农民桑红杰就会坐在电脑前,开始一天的工作:种蔬菜。

桑红杰租种的农田位于杭州湾大桥南岸的慈溪掌东镇,离家约有10多公里。可他每天只需在家里打开电脑,点点鼠标,就能轻松管理90多亩蔬菜。桑红杰加盟的慈溪市蔬菜开发有限公司,不久前刚刚开发了一套数字远程信息系统,正是凭借该系统,桑红杰当上了"数字农民"。与他一样,慈溪已有100多户农民用上了该系统。

图 1-1 《新民晚报》2008 年 4 月 1 日的新闻报导

在人类社会早期的农业和工业经济时代,土地等自然资源是占支配地位的生产要素,直接或间接地利用自然资源是社会经济存在和发展的基础。而信息时代是直接利用信息资源,通过对信息进行数字化、程序化加工处理,使信息成为一种独立的、直接的生产要素,并生产出以知识密集为特点,具有高效使用价值的信息产品。在信息产业中,信息资源成了首位的生产要素。信息产品正日益丰富,在国民生产总值(GDP)中的比重不断扩大。2007 年我国信息产业销售额为 5.6 万亿元人民币,信息产业增加值占 GDP 比重达到 7.9%。当年我国信息产业规模已超过日本,连续四年仅次于美国位列世界第二,并已经成为我国第一大支柱产业[5]。从国家机构设置来看,早在 1998 年 3 月,第九届全国人民代表大会第一次会议通过了《关于国务院机构改革的决定》,在原有的邮电部、电子工业部及广播电影电视部、航天工业总公司、航空工业总公司的信息和网络管理政府部门的基础上设立信息产业部。2008 年 3 月,第十一届全国人民代表大会第一次会议通过了《国务院机构改革方案》,将发改委工业行政管理的有关职责、国防科工委管理核电以外的职责、信息产业部和国务院信息化工作办公室的职责,整合划入工业和信息化部[6]。信息工业在我国国民经济领域将扮演更加重要的角色。

在信息时代,该如何理解"信息"呢?从更广泛的角度来看,应先讨论对知识、真理和信息已有的两种基本观点。我们相信客观世界,包括自然世界和人本身及其相互之间的关系是人类追求知识和真理的最直接来源和动力,虽然这并不表示就一定能够正确地理解、掌握客观世界规律。对世界的不同认识就产生了各种不同历史时代的认识观。文艺复兴时期意大利画家拉斐尔(Raphael Sanzio,1483—1520)的名画"雅典学院"(School of Athens,图 1-2)中的两个中心人物,柏拉图和亚里士多德的动作就可以形象地刻画出人们对客观世界认识的

原则性差别[8]。右边的亚里士多德的手指完全张开，指向地上的事物，他看重的是地上事物存在和发展的各种客观规律，他并不关心超越于万物之上是否存在大统一的真理及其永恒价值。这就好像在物理学中爱因斯坦毕生所追求却没有实现的统一场论对人们的日常生活来说还没有太多的现实意义一样。创建系统的逻辑推理理论及其各个科学分支体系是亚里士多德的一大成就。相反，柏拉图却用一个手指指向天，他或许更看重上天的启示真理、未来的生活理念和终极关怀。即使遇到现实生活中不合理的价值体系或者不公平的现象，也不能放弃对绝对真理的热心追求和向往，并持守随之而来正常的价值体系和生活方式。这两个对立的手势，很形象地描绘了他们思想上的原则分歧及其这种分歧在艺术、哲学和自然科学领域的普遍意义，尽管他们各自都拿着代表着知识的书本。如果可以把知识作为信息所表达的内容的话，那么在 1.1.2 节我们可以知道信息的表达形式和信息所表达的内容两者之间是可以相互独立的。这就是说，在对信息进行获取、存储、变换处理和通信时，我们并不关心它所表达的具体内容。对于计算机和其他数字型微处理器来说，信息一般都是用“0”和“1”进行符号化表示，它所表示的内容还需要解释程序进行转换处理。以微软公司的 Word 格式文档为例，它所保存的文档在存储器内部就是一长串的“0”和“1”符号，一般称做“文件”。只有用 Word 文档处理程序才能把这种文件转换为具体的文档及其版面格式。在 Windows 系列的操作系统中，文件的格式一般需要扩展名来体现，例如 Word 文档的扩展名为. doc 或者. docx。而在基于 UNIX/Linux 的操作系统中，文件扩展名和文件内容之间并没有格式匹配要求，从而隔离了文件名和文件内容之间的联系。从客观规律认识的角度来看，信息和它所表达内容之间的独立性背后所隐含的世界观比较符合亚里士多德的思想——即追求形式或者逻辑上的一致性或者无矛盾性，而不是很关注隐藏在形式或者逻辑系统后面的价值。这种形式和内容相对独立的思想比较可能演变为相对主义和悲观论[4]，从而容易让人们失去对客观世界和人及社会本身的终极关怀。我们所处的信息时代，相比于历史上的其他时代，例如文艺复兴和思想启蒙时代来说，更可以称为“虚拟化时代”或“技术理性时代”，即我们在知识的研究、文明的探索和建设过程中，更强调的是它们的一致性、无矛盾性，而不是它们所表示内容的完备性或客观实在性及其价值，特别是当我们只能定义事物之间的相对关系，而难以定义各个独立事物存在时的“客观性”的时候。信息技术的发展加剧了一致性和客观真实性两者的割裂[7]，这种割裂关系可以通过数字电路设计理论——布尔逻辑中的正负逻辑来清楚地理解。

图 1-2　名画“雅典学院”中心人物柏拉图和亚里士多德[8]

英国数学家布尔(George Boole，1815—1864)充分认识到人们在科学研究或者是日常生活中普遍存在的逻辑思维错误及其造成的后果，因此他引入了一套严格的符号系统并提出了严密的逻辑推理理论——布尔逻辑[9]。也有人把他的理论称做符号逻辑、数理逻辑、现代逻辑、逻辑代数等。这套推理理论和古典的亚里士多德逻辑学迥然不同——它利用严谨的数学方法研究抽象思维的一致性。如果一种表述(或者更确切地说是一个命题)正确地描述了一个客观事实，那么我们就判断它为“是”；反之为“非”。根据布尔逻辑的原则，我们首先把日常生活中用于判断正确性的“是”和“非”分别用

“1”和“0”进行符号化；然后再定义一些基本的逻辑运算，如“与”、“或”、“非”和一些进行逻辑演算和推理的规则、定理。根据布尔逻辑的规定，一个命题不可能同时为“是”和“非”，因此命题之间的推理关系就等价于严密的符号运算或者命题演算，从而在理论上避免了错误的是非判断。虽然布尔逻辑只能进行二值运算，远比中国人发明的具有混合进制的算盘技术要简单，但是从机器实现的自动化程度来看，布尔逻辑更容易实现。下面举例说明布尔逻辑的严密性，并同时从正逻辑和负逻辑的观点阐述逻辑一致性和真实性的区别与冲突。

例 1-1 考虑到环保和健康的绿色低碳要求，小潘事先承诺：“如果明天不下雨，我就骑自行车上班。”事后证明那天没有下雨，但是小潘也没有骑自行车上班。基于这样简单的事实，我们很容易判断小潘讲话不算数或者没有兑现承诺。对于这样一个简单的命题推理，我们可以借助布尔逻辑来判断小潘是否讲话不算数。小潘这句话包含两个子命题，分别用 p 和 q 表示如下：

p：明天下雨；q：小潘骑自行车上班

首先我们假设“是”和“非”分别用符号“1”和“0”表示，这就是我们习惯的正逻辑规则。根据事后的客观事实：“那天没有下雨”、“小潘没有骑自行车上班”，我们可以知道两个子命题的值：$p=0, q=0$。根据正逻辑的规则，小潘的话就可以符号化为“$p+q$”，其中“+”为“逻辑或”运算。由于 $p=0, q=0$，所以 $p+q=0$。又由于“0”表示“非”，所以小潘的话属于“非”命题，或者说，小潘没有兑现他的承诺或说他讲话不算数了。

现在再使用另一种规则，假设“是”和“非”分别用符号“0”和“1”表示（刚好和正逻辑的“是”、“非”符号化关系颠倒过来），这是我们习惯上不用的负逻辑。根据客观事实，我们得到两个子命题均为“非”命题，即：$p=1, q=1$。根据负逻辑的规则，小潘的话符号化为“$p \cdot q$”，其中“·”为“逻辑与”运算。由于 $p=1, q=1$，所以 $p \cdot q=1$。根据负逻辑的规则，“1”表示“非”。因此小潘的话同样属于“非”命题，或者说，小潘没有兑现他的承诺。这和正逻辑所得到的结果相同。

从例 1-1 可以看出，布尔逻辑本身并不知道“0”表示“是”还是“非”。换句话说，可以用正逻辑，也可以用负逻辑进行正确的逻辑推理，只要在推理过程中保持一致的规则，两者得到的结论是相同的。这也同样可以说明拜廷格提出的第一个信息的特征：无须知道确切的含义就可以对信息或数据进行传输和处理[10]。在有关信息的逻辑推理中，人们关注的是一致性，而不是信息形式和信息内容对应关系的正确性或合理性。实际上，在布尔逻辑中，任何命题表达式都存在一个对偶式：只要把命题中的“与”和“或”、“0”和“1”进行置换就可得到。在逻辑层次上，一个命题和它的对偶式都是没有矛盾的，且可以很方便地转换——看起来好像是“是”、“非”颠倒的形式。但是一个命题和它的对偶式都不能断定它和它所表达的客观事实之间对应关系的合理性。换句话说，布尔逻辑只是符号系统，它不能断定这个符号系统和它所表达内容的对应关系的“正确性”。因此，信息技术的发展加剧了信息推理的一致性和信息内容的“完备性”之间的割裂（这种“完备性”包含了符号化的形式系统及其内容之间的对应关系），甚至有人会否认客观真理的存在——把客观世界也看成是一种符号化的游戏规则而已。关于“完备性”或译作“完全性”（completeness）的定义和这种割裂关系的讨论已经超出了本书的范围，有兴趣的读者可以参考文献[7]。这种割裂关系发展到一定程度就引发“虚拟现实”的思想，从而导致“相对主义”或“虚无主义”——每个人只要活在自己以为是正确，而实际上只是一致的世界中；根本不在乎这个世界和客观世界的对应关系是否

合适。在这一点上，我们认为是非的对应关系是不能颠倒的。表面上看来，用“0”表示“是”或“非”只是一种逻辑假设而已。但是在实际应用中，它们并不像正负逻辑所具有的对偶性。例如电脑的关机和开机在实际应用中不具有对偶关系。这是由于电脑开机时可以告诉用户它是开机的；而在关机时却无法告诉用户它是关机的。当然可以假设电脑的默认状态为关机，即在没有收到电脑发出的信号时默认它是关机的。人的睡眠和清醒状态两者也具有类似的属性——睡眠时的人不可能告诉别人他正在睡眠。再进一步从实际情况来看，一个真正酒醉的人不会告诉你他是酒醉的；相反他都会说自己没醉。这些例子都说明了在实际应用中正负逻辑对应关系的对偶性并没有简单的普适意义。

对于数学和逻辑学的真实性以及相关的完备性问题，1930 年前的科学家们都对数学和逻辑充满希望，像大数学家希尔伯特（David Hilbert，1862—1943）所提出的公理化系统一样。他相信只要不断扩展公理化系统，那么世界上所有的不确定性问题都会迎刃而解。1931 年，可称做是自亚里士多德以来最伟大的逻辑学家哥德尔（Kurt Godel，1906—1978）发表了不完全性定理，或译作不完备性定理。该定理明确指出了任何符号系统要么是有矛盾的，要么是不完备的，在一致性和完备性两者之间不能兼得[12]。他的定理使得许多科学家的梦幻破灭，例如冯·诺依曼（John von Newmann）一开始从事数学领域的研究，并取得了诸多开创性的成果；在哥德尔发表了不完全性定理后，他决定放弃数学，而转去从事计算机领域的工作，后来斯坦福大学在官方论文集称他为“计算机之父”。

信息时代也只是表明处理信息的能力和效率在突飞猛进，而不能说我们就掌握了所有的知识或者说信息所表示的内容就一定是正确的。我们依然生活在一个充满矛盾、充满挑战、充满变化的客观世界中，不能局限于已经掌握的知识，也不要沉溺于信息时代可能会滋生的相对主义和所创造的“虚拟现实”或虚幻的感觉之中。哥德尔的不完全定理并不能让我们停止对未来美好的追求。笔者曾经问过学生这样一个问题：你来学校学习的主要目的是什么？所听到的答案基本上是“为了能够找一份好工作，即使我不喜欢这个专业或者是目前学得很辛苦也是愿意的”。对于在学习生活中所学到的各种知识和随之而来的更高的价值观追求则缺乏必要的乐趣和奋斗目标。大学不同于一般的技术学院或职业学校，它不应该和社会的就业需求严格同步。相反，大学的学科和专业设置应该能够超越不断波动的社会就业需求变化。如果大学失去了对学术价值孜孜不倦的追求，那么就成了普通的技术培训学校，这和大学发源的初衷大相径庭。更严重的是，学生以找工作为目标的大学生活会埋没一代又一代的莘莘学子[13]。2009 年 7 月 2 日，《人民日报》刊登了一篇报道，明确指出：大学不能没有“精神围墙”。大学所要做的就是把思想和文化系统化、规范化，将其转变为课程，再把它们传授给学生。报道中引用了美国教育家德怀特·艾伦（Dwight W. Allen）的一句话：“如果我们使学生变得聪明而未使他们具备道德性的话，那么我们就在为社会创造危害。”这从另一个角度说明了，传授专业知识固然重要，但培养一个人的道德精神却更加重要。另外，有兴趣的读者可以阅读由企鹅出版社出版，获普利策文学艺术奖的 Douglas R. Hofstadter 先生的著作 *Godel, Escher, Bach: an Eternal Golden Braid*。这是一本杰出的科普名著[14]，笔者曾经阅读过其英文版的部分章节，十分佩服作者的思维能力、想象力和孜孜不倦的追求精神。从人类文明发展的历史来看，现代美国著名史学家 Will Durant 在他的巨著《世界文明史》中描述希腊罗马历史时就提醒我们[15]：“希腊那个时代，产生了很多这样的人：有知识而无道德，有能力而无顾忌，有勇气而无忠贞。”如今在信息时代，我们更

要做一个有知识、有道德、有热情、有理想的健全人。

在理解了信息和它所表达的内容之间的独立关系后，人们就不再关心信息所表达的具体内容，而只关注信息的变换和处理技术，后者是信息时代来临和发展的原动力。信息时代的来临，是以半导体集成电路、计算机及其软件和信息化的网络技术为必要条件，以成熟的信息存储、处理和通信技术为技术背景，只有文字和活字印刷术这些前提条件还不能够形成信息时代。因此下面回顾一下信息时代来临的技术前提和历史背景。如图 1-3 所示，基于对客观世界的研究热情，亚里士多德开启了古典逻辑学的大门，为后续对客观世界的正确理解和逻辑推理奠定了扎实的基础。1854 年，布尔发表了《思维定律研究》一书，为数字电路的设计和优化提供了数学理论基础[9]。1938 年，香农（或译申农，Claude Elwood Shannon，1916—2001，照片见图 1-4）在麻省理工学院获得电气工程硕士学位，他的硕士论文题目是《继电器与开关电路的符号分析》[16]。香农的开创性工作在于：布尔代数中的“0”或“1”可以对应于继电器中的两个状态——“开”或“关”，并且布尔代数中的各种逻辑运算都可以通过继电器的开关控制来实现，使得用机械设备或硬件电路构造的机器具有快速、高效、持久的计算和推理能力。当然，基于例 1-1 对正逻辑和负逻辑的理解，我们可以把布尔代数中的“0”或“1”和继电器中的两个状态（“关”或“开”）之间的对应关系倒过来。不管“0”表示“开”或者“关”，只要是确定的对应关系就能够实现逻辑上没有矛盾的符号计算或逻辑推理。由于正逻辑和负逻辑实现同一个逻辑函数时的逻辑表达式是不同的，因此它们所对应的硬件电路成本和性能一般也会不一样。基于这样的思想，就可以把布尔代数的逻辑推理过程，或者可以说是人脑的思维过程，用各种继电器的开关组合操作来优化实现。这些逻辑开关的基本组合就叫做基本逻辑运算硬件单元或逻辑门。这样看来，我们就赋予了这些逻辑开关一定的数学计算或推理能力。或者也可以说，人类的思维过程可以借助继电器组成的机器来实现——让机器模拟人类的思维过程。这是一项革命性的成果。现在我们学习数字电路设计技术时，都要学习“香农定理”（或译作“申农定理”）来展开或分解复杂的布尔表达式并进行逻辑化简。1992 年当美国杂志 *Scientific American* 的编辑 John Horgan 采访香农时，他富有深意地说到：“我始终热爱这个词——布尔。”这篇带有香农亲笔签名和照片的论文发表于 1992 年的 *IEEE Spectrum* 杂志[17]。

但是用继电器作为开关实现硬件电路的效率还是非常低的。随着电子管技术和半导体集成电路技术的发展，人们先后用电子管的开关状态、双极性晶体管和目前主流的 CMOS（complementary metal oxide semiconductor）晶体管取代继电器的机械开关状态来实现布尔代数中的“0”和“1”。这样复杂的逻辑思维或者是信息处理过程就可以很方便地集成到一小块芯片中去，从而极大地提高了信息处理的效率，为后来通用计算机的核心——CPU（central processing unit，中央处理器）芯片设计与实现奠定了硬件基础。在通用计算机理论方面，1936 年阿兰·图灵（Alan Tuning）提出了通用的计算模型，他以一种抽象的方式详细地描述了可编程计算机的概念，并且证明了基于一个通用机器（后来称为“图灵机”）就可以确定一个算法的步骤和任务完成方式。任何可以由数字电路执行的算法都存在基于通用图灵机上的等价算法来完成相同的任务。图灵机理论为研究计算的复杂性和可计算性等问题提供了重要的理论基础。图灵机的理论模型、非继电器开关的硬件电路设计基础和第二次世界大战期间对大量数据计算的需求共同促成了电子计算机的诞生。1945 年冯·诺依曼等人总结并发表了“存储程序计算机”（stored program computer）的概念，即计算机运行

时所需要的指令流可以通过数据存储的方式由计算机自动读取执行，而不一定要用手动控制开关或者纸带打孔的方法输入可执行程序。虽然在过去的半个世纪内，CPU 的集成度和处理速度指数上升，但是 CPU 的基本架构还是遵循冯·诺依曼结构。在以后的讨论中我们可以发现，不管是 CPU、GPGPU(general purpose graphic processing unit，通用 GPU)、DSP 还是 PLD(programmable logic device，可编程逻辑器件)和 FPGA(field programmable gate array，现场可编程门阵列)，它们的可编程性和灵活性都是通过不同粒度的存储单元来实现的。

基于冯·诺依曼体系结构的通用计算机结构的重要性在于人们可以很方便地完成各种复杂的信息处理和变换。为了进一步满足计算效率和计算机之间通信的需求，在冯·诺依曼发表“存储程序计算机”三年后的 1948 年，贝尔实验室的香农又发表了《通信的数学原理》等一系列论文[11]。这篇论文为实现计算机网络通信打下了坚实的理论基础，香农也因此被冠为“信息论之父”。如图 1-3 所示，以下这三个必要条件：成熟的半导体集成电路的设计和制造技术、高效的通用计算机软硬件平台以及无处不在的计算机通信网络促成了信息时代的到来。

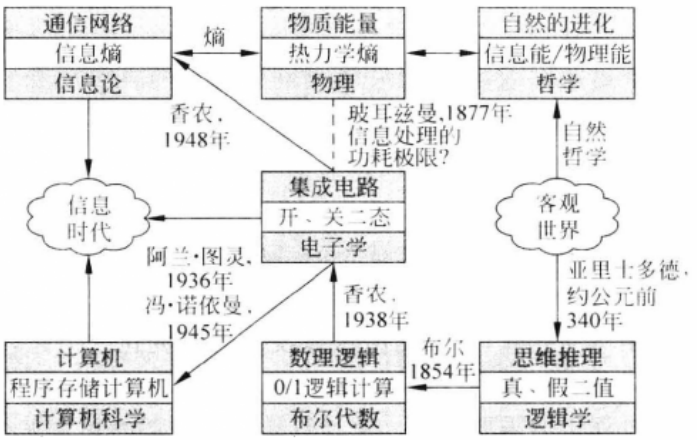


图 1-3　信息时代的来临

从计算机到互联网，从有线电话通信网到数字电视、移动数字通信网，信息时代的发展给我们的日常生活带来巨大的变化。物联网可能是网络时代的下一个发展领域。它把信息处理和交换的领域从计算机、电视和电话进一步扩展到所有的商品或物品，只要在物品上嵌入电子标签或条形码用于信息存储，所有的物品所组成的物联网就可以和已有的各种网络结合在一起，实现了人和人、人和物之间多渠道的信息沟通。从推动经济发展角度来讲，物联网可能会成为信息时代的下一个发展途径。

但是，任何事物的诞生和发展都不是永无止境的。随着半导体集成电路的集成度和处理效率不断提高，在过去的近半个世纪内，作为计算机核心部件 CPU 的集成度基本上符合摩尔定律指数增长，尽管增长的指数常数有所修改。但是随着信息处理对超大规模集成电路提出的需求，芯片的功耗问题日益突出。2004 年，Intel 公司由于功耗原因取消了单核新一代 Tejas 处理器，从而进入了双核或多核处理器时代。近几年对于信息变换在理论上是否一定需要物质或能量的问题展开了热烈的讨论。图 1-3 右上角列出了信息和自然哲学的

关系。美国普林斯顿大学的 John A. Wheeler 教授就提出物理世界是由信息构成的观点，他指出信息才是自然界进化的重要动力，物质和能量不过是附属物而已。对于这方面的研究内容感兴趣的读者可以阅读文献[22]。

香农最初是用继电器来表示信息的基本状态，对继电器的控制肯定需要能量。目前普遍使用的是更可靠、精细、易集成的晶体管开关，开关状态的控制就有电容的充放电过程，这也要消耗能量，尽管单位电容的容量随半导体工艺的更新而不断减小。如图 1-3 所示，针对物质和能量问题，早在 1877 年玻耳兹曼就提出了热力学中“熵”的概念，用于描述大量分子或其他粒子的概率特性或无序性。在热力学第二定律中，他对于熵概念做出了统计力学意义的定义：

$$S = k \cdot \ln m \tag{1-1}$$

其中，k 表示玻耳兹曼常数——1.38×10^{-23}J/K；m 表示微观粒子的状态数目。根据文献[10]中第 3 章和第 8 章的计算，由热力学第二定理可以得到处理 1 位二进制信息所需的最小能量 W 为

$$W = 2kT \cdot \ln 2 \tag{1-2}$$

其中，T 为绝对温度。这样看来，虽然目前半导体工艺控制 1 位状态所需要的能量比式(1-2)要高出好几个数量级，但是从理论上来说信息变换过程好像是一定需要能量的。另外，从信息论的角度来看，布尔代数中的基本运算，如常用的二输入“与非”门都是不可逆变换，即不可能从一位输出状态恢复两个输入值。这种输入输出之间熵的变化就需要消耗能量。假设输入输出是可逆关系，是否意味着不需要消耗能量的信息变换过程？对于图 1-3 所示的“信息处理的功耗极限”问题，近几年在量子计算领域的研究指出量子计算是酉性可逆变换。量子信息处理和变换本身是不需要能量作为代价的。例如 2003 年 Nature 杂志发表了全光学方法实现的量子受控非门(CNOT 可逆门)[20]。从信息论的角度来看，量子计算过程是不需要消耗能量的，因为它是可逆计算，输入输出状态没有熵的变化。但是，量子计算结果的测量过程是不可逆的，量子状态一旦测量，就从原来的叠加态塌缩到基态，从而引起了信息熵的变化，因此量子测量过程仍然需要能量。从计算效率和功耗的角度来衡量，相信量子计算的研究必定为信息处理和变换技术开辟崭新的道路。

最近几年由于量子力学中的叠加态和纠缠态性质，使得量子计算具有超乎寻常的并行计算功效。它为解决利用传统计算机难以解决的 NP-hard(non-deterministic polynomial，非确定性多项式)问题提供了多项式复杂度算法，其中比较著名的是 Shor 先生于 1994 年发表的在量子计算机上分解因数的快速算法。此算法可以用于破解广泛使用于电子商务加密系统的 RSA 密码。按照当时的条件，破解 RSA129 密码大约需要 1000 台工作站计算 8 个月的时间；根据 Shor 的计算结果，假设量子计算机的时钟频率为 100MHz，则只需要几秒钟的时间。另一个著名的量子算法是 IBM 公司的 Grover 先生于 1996 年提出的数据库快速查找算法，它使时间复杂度从经典方法的 $O(N)$ 降低到 $O(\sqrt{N})$。据英国广播公司(BBC)于 2009 年 9 月 4 日的报道，英国科学家已经成功研制了光学量子计算芯片原型，可以运行 4 位数据的 Shor 算法[19]。从社会和国家的安全保密性来考虑，各国政府花很大的人力和财力用于研究和开发量子计算理论和物理实现技术。量子计算的研究成果将对推进信息技术的发展具有划时代的意义，对国计民生和金融安全技术以及国防建设具有重大的现实意义和广阔的应用前景[21]。

1.1.2 信息的度量与变换处理

作为信息时代的主体——信息，目前还没有一个公认的定义。但是从1.1.1节关于信息时代的特征可知，信息和它所表示的内容具有独立性。相对于信息所表示的内容来说，我们更关注于符号化后二进制信息的处理算法和变换效率。从这个角度来看，采用数学方法定量描述信息更好。更确切地说，根据信息所表示内容的概率属性来度量信息。关于信息的一般特征，文献[10]列出了由U. Baitinger提出的4个重要特征和技术需求。

(1) 信息和它所表达的内容及其通信方式相对独立：由于信息是通过数学的方法进行抽象描述的，因此无须知道信息所表达的确切含义就可以对信息或数据进行传输和变换处理。从信息处理的技术层面来看，我们所关注的主要是符号化后的各种字符信息或二进制信息，而不是它所表现的真实内容。这个特征指出了信息和信息所表达内容的区别，同时表明信息技术具有广泛的适用性和通用性。例如一封电报、一条手机短信和一张照片，一场体育赛事结果，一个网站的网页或一条微博的内容，数学中的一个定理，晶体管的开关状态，电子的自旋状态，抑或是细胞核染色体中的DNA(deoxyribonucleic acid，脱氧核糖核酸)都包含着一定量的信息。在后面的介绍中我们可以发现对这些信息能够进行数学度量，尽管它们表示各种不同类型的内容。因此信息技术主要是指对它们进行提取、编解码、存储、变换处理和传输等过程，其本身并不局限于某一个具体的应用领域。在通信领域中，信息的具体表达形式可称做“消息”(message)。这种相对独立的处理方法就很符合符号化后的信息和信息所表示内容之间的相对独立性。

(2) 信息存储需要某种类型的载体：物质或能量。由以上第一个特征可知，信息是一种数学描述，其本身不是一种物质或能量。但是对信息进行存储时，就需要物质或能量载体。对于目前十分普及的数字信息来说，移动硬盘或者便携式硬盘都是很方便的信息载体，对这些信息的读取和写入都需要一定的功耗或能量。在没有载体的情况下，信息能否存在就是如图1-3中的自然哲学问题。这个特征表示在现有的条件下，信息的存储和变换处理均需要消耗一定的物质或能量。

(3) 信息在各种载体之间的无损交换：各种信息载体可以进行信息交换而不会丢失任何信息所表达的内容。这个特征表示信息可以在不同的载体之间进行无损交换，例如计算机内存的信息内容可以转存到外部存储器、移动硬盘或者是DVD等存储格式，再通过以太网或无线网络以数据包的形式发送出去。

(4) 高效的信息传输和处理技术：信息交换可以不损失任何信息内容，但是信息借助于某一种载体在传输过程中可能会出现失真或损耗。因此，通过数学抽象方法进行高效、安全又无任何丢失的信息传输和处理是对信息技术提出的挑战。为了提高信息传输的效率和安全性，目前已有各种信源压缩、信道纠错编码、信息保密技术等研究领域。在通信领域，基于光子的光纤通信效率，如速度和带宽等性能要优于基于电子的电磁波通信效率。

通过以上各点，可以理解信息的一些基本属性和关键技术需求，但是还没有很好地回答“什么是信息?”这个基本问题。不过从信息处理的角度来看，我们更关心的是如何对符号化后的信息进行有效的数学处理和数字通信。因此信息论之父——美国科学院、工程院院士香农利用数学方法给予了很好的参考定义：信息是事物运动状态或存在方式的不确定性的描述。图1-4是从网站 http://it-science.net/images/shannon.small.jpg 下载的香农年轻

时的照片。1948 年，他发表了《通信的数学理论》[1]。该论文从数学的角度，利用概率的方法定量地描述了信息，并解决了信息通信中的一些基本问题。他的一系列论文奠定了现代信息论的基础。另外他在数字逻辑开关电路设计、计算机和自动控制、人类遗传学等多个领域都颇有建树。下面根据以上关于信息的第一个基本特征，首先讨论如何定量描述信息，然后再简单介绍通信系统中信息的存储、交换、传输和处理等技术。1.1.3 节开始介绍用于信息处理的半导体集成电路技术。

图 1-4　信息论之父香农

香农运用数学中随机事件的不确定性来定量地描述信息。例如天气预报说“明天要下雨”，这表示如果天气预报正确的话，“明天下雨”这个事件发生的概率是 100%；而在得到这个信息以前，明天下雨、晴天等各种天气都是不确定的。因此信息所提供的信息量和它所描述的事件概率 $p(x)$ 直接相关。一个事件或俗称一条信息 x，所包含的信息量称做自信息量 $I(x)$，它应该满足以下条件[23]。

(1) 发生概率小的事件对应的信息量要大于发生概率大的事件，即 $I(x)$ 是 $p(x)$ 的单调递减函数。从直观的意义上说，如果一个事件的发生概率为 1，即它描述的是完全确定、必然发生的事件，那么它的自信息量为 0，即对于肯定要发生的事情的描述并不能带来任何信息量，即当 $p(x)=1$ 时，$I(x)=0$。如果一个事件的发生概率为 0，那么它的自信息量为无穷大，即对于肯定不会发生的事情，它一旦发生所带来的信息量为无穷大，即当 $p(x)=0$ 时，$I(x)=\infty$。

(2) 自信息量应满足线性关系，即对于两个独立的事件，它们所对应的信息量应该等于各事件自信息量之和。

根据以上条件，可以推测对数函数满足上述要求。因此事件 x 的自信息量 $I(x)$ 被定义为

$$I(x) = -\log p(x) \tag{1-3}$$

其中，$p(x)$ 表示信息所表示内容出现的概率。容易验证对于两个独立的事件，它们共同发生的概率是两者的乘积。因此可以得到

$$I(x) + I(y) = -\log p(x) - \log p(y) = -\log(p(x)p(y)) \tag{1-4}$$

上式表示条件(2)得到满足。

目前半导体集成电路技术采取的大多是二值逻辑，因此式(1-3) 中的对数底数为 2。信息的单位是比特(bit，即 binary type 的缩写)。如果对数底数为 e 或者 10，那么信息量的单位分别为奈特(nat，natural type)或笛特(det，decimal type)。

例 1-2　假设上海市某年度 2 月份的天气可以分为“晴天”、“阴天”、“下雨”和“下雪”四种可能情况，分别用 x_0、x_1、x_2、x_3 表示，它们的概率分别为 50%、25%、12.5%、12.5%。根据式(1-3)，它们的自信息量分别为

$$I(x_0) = -\log p(x_0) = -\log 0.5 = 1\text{bit} \tag{1-5}$$

$$I(x_1) = -\log p(x_1) = -\log 0.25 = 2\text{bit} \tag{1-6}$$

$$I(x_2) = -\log p(x_2) = -\log 0.125 = 3\text{bit} \tag{1-7}$$

$$I(x_3) = -\log p(x_3) = -\log 0.125 = 3\text{bit} \tag{1-8}$$

应注意，式(1-5)～式(1-8)中的“比特”是信息量的单位，自信息量可以是整数，也可以是小数。从通信的角度来说，以上所定义用于衡量信息量的概念后来通常就被称为香农熵或信息熵。也就是说，一条消息的香农熵就是用二进制编码这条消息所需的最少位数，这个位数可以是小数。如果一条消息对应的概率越小，那么它包含的信息量就越大。

自信息量是描述具体事件或符号化后具体一个符号的信息量，它还不能表示发生这个事件的信源的信息量。为了能够度量一个信源本身的信息量，可以用该信源能发生的所有事件的数学期望来衡量。它也称为信源的平均自信息量、平均信息量或平均信息熵，从而达到描述信源所包含各种信息的总体效果的目的。

$$I(X) = \sum_{i=0}^{n-1} p(x_i) I(x_i) = -\sum_{i=0}^{n-1} p(x_i) \log p(x_i) \tag{1-9}$$

其中，n 表示该信源所发出的事件个数或符号化后的符号个数。例如有两个信源 X 和 Y，它们各包含两个字符信息——x_1、x_2 和 y_1、y_2，相应的概率空间分布为

$$\begin{bmatrix} X \\ P(X) \end{bmatrix} = \begin{bmatrix} x_0 & x_1 \\ 0.99 & 0.01 \end{bmatrix}; \begin{bmatrix} Y \\ P(Y) \end{bmatrix} = \begin{bmatrix} y_0 & y_1 \\ 0.5 & 0.5 \end{bmatrix} \tag{1-10}$$

那么它们的平均信息熵分别为

$$I(X) = -(0.99\log 0.99 + 0.01\log 0.01) = 0.0808\text{bit} \tag{1-11}$$

$$I(Y) = -(0.5\log 0.5 + 0.5\log 0.5) = 1\text{bit} \tag{1-12}$$

因为对于 X 信源来说，99%的概率会出现 x_0 符号，而只有 1%的概率会出现 x_1 符号。因此它所提供的信息量不如信源 Y 多。式(1-11)和式(1-12)的计算结果可以说明：等概率分布时所提供的信息量最大。从数值上来说，平均自信息的数学表达式和统计物理学中的热熵表达式很相似，因此平均自信息量也叫做平均信息熵。例如可以把汉字字库中的各汉字及其使用到的平均概率利用式(1-9) 计算，可以得到每个汉字的平均信息熵。根据联合国的统计，汉字字符的平均信息熵是 9.65bit，是当今世界各种文字中最高的。英语中的基本字符包括 26 个字母和一个空格，它们的平均信息熵为 4.03bit。

因此计算机处理汉字比处理英语字符要复杂得多。汉字必须采用双字节的 Unicode 码，其中每个字节为 8bit；英语文本字符用 8 位的 ASCII(America Standard Code for Information Interchange)码表示就够了。这种二进制字符长度和平均信息量的量化关系可以通过本章习题 2 很方便地证明。

在理解了信息的度量和信息变换处理的字符串长度关系后，我们来讨论平均信息熵和信息通信中码长，即信源编码后的长度的关系。基于香农在《通信的数学理论》一文中提出的通用通信系统模型，目前信息通信系统的基本流程如图 1-5 所示[23]。贯穿整个系统流程的就是信息流，它主要包括对信源信息的提取、存储、变换、传输和处理等技术。为了保证信源和信宿之间正常的通信，必须采用一定长度的二进制数进行信源编码和译码。同时为了提高通信的安全性和抗干扰性，从而消除信道中噪声的影响，还要引入信道编码、信道译码和加密解密等处理技术。香农第一编码定理就确定了无噪声通信时信源编码平均长度的下限，它指出编码后的平均码长不能小于平均信息熵。我们可以通过以下示例说明。

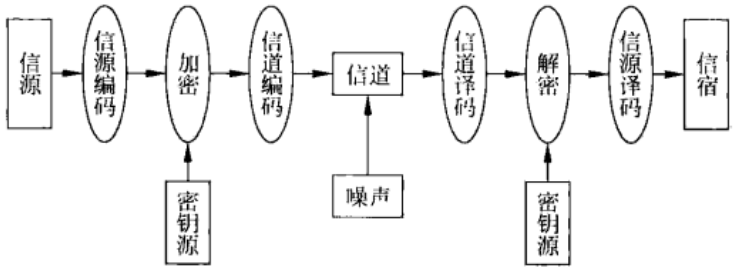


图 1-5 信息通信系统模型[23]

例 1-3 设某班级有 n 个学生的期末考试卷面成绩分布为：50%为 A，25%为 B，12.5%为 C，12.5%为 D。现在要把所有学生的成绩以最短的二进制编码信息发送出去，问如何对 A、B、C 和 D 四种成绩进行二进制编码？

解：按照我们所熟悉的也是最简单的编码方法，A、B、C、D 共有 4 个字符，因此可以用两个字符对它们进行编码。例如，A、B、C、D 分别用 00、01、10、11 来编码。那么 n 个学生的成绩就可以按照学生名单的次序发送出去，总共需要 $2n$ 个字符，即每个成绩的平均码长为 $2n/n=2$。

现在再问这样一个问题：按照这种编码方法发送的字符串长度是否为最短？这个问题可以利用平均信息熵来解决。根据成绩分布可以求得平均信息熵为

$$\begin{aligned}\sum_{i=0}^{3} -p(x_i)\log p(x_i) &= -(0.5\times\log 0.5+0.25\times\log 0.25\\ &\quad +0.125\times\log 0.125+0.125\times\log 0.125)\\ &= 1.75\end{aligned}$$

根据香农编码定理，我们有可能找到一种编码方法，使得编码后的每个成绩的平均码长小于 2。考虑到成绩不是等概率分布，因此可以采用霍夫曼(Huffman)编码方法，将出现概率大的成绩用较短的字符编码，出现概率小的成绩用相对较长的字符编码。例如，A、B、C、D 可以用 0、10、110、111 来编码。根据前缀码的性质，“这种编码后的字符串”可以唯一确定各学生的成绩。假设每个成绩的码长用 $l(i)$ 表示，这样每个成绩的平均码长为

$$\sum_{i=0}^{3} p(x_i)l(x_i) = 0.5\times 1+0.25\times 2+0.125\times 3+0.125\times 3 = 1.75$$

根据香农编码定理，此时平均码长已经等于平均信息熵。因此从码长的角度来看，这是一种最优的编码了；或者说，不存在其他的编码方法，可以让每个成绩编码后的平均码长小于 1.75。

1.1.3 半导体技术和数字集成电路的发展

从信息时代的特征和 1.1.2 节对信息变换处理的需求来看，编码后的数字化或符号化信息主要需要三种核心技术：存储、变换处理和通信。这三种技术最有效的实现途径是基于硅材料的半导体集成电路，其核心器件分别对应于存储器、各种微处理器和网络路由控制器。因此，信息时代的核心技术离不开半导体集成电路。这三种器件的核心作用同样体现在数字芯片的内部架构。数字集成电路内部的关键要素是：触发器或者寄存器等存储单

元、计算单元或者逻辑门单元和总线或互连资源。例如通用的 CPU 内部一般包含不同存储粒度的寄存器和高速缓冲器、ALU(arithmetic logic unit,算术逻辑单元)单元和总线结构或其他互连结构等。FPGA 芯片内部也是包含有配置信息的 SRAM(static random access memory,静态随机访问存储器)存储单元、触发器、嵌入式存储器核、各种基于 LUT(look-up table,查找表)的可编程逻辑单元和用于传递信号的层次式互连资源。

从历史的角度来看,最早在工程上把人类的逻辑思维过程,或者说是通用信息处理过程用硬件办法来实现的里程碑还要归功于香农。虽然中国人很早就发明了混合进制的算盘,但是它主要应用于基本的数字运算或者生活中的账务计算,还不能实现通用的逻辑计算和复杂的信息处理过程。1938 年香农在 MIT 获得电气工程硕士学位,硕士论文题目是 *A Symbolic Analysis of Relay and Switching Circuits*("继电器与开关电路的符号分析")。当时他已经注意到电话交换电路设计与布尔代数之间的类似性,即把布尔代数的"真"与"假"和机械系统的"开"与"关"对应起来,并用"1"和"0"表示。于是他用布尔代数分析并优化开关电路,这就奠定了数字电路的理论基础。这篇论文曾经被认为是"这可能是本世纪(当时是指 20 世纪)最重要、最著名的一篇硕士论文"。当时还没有半导体晶体管,只能用继电器的开关状态来实现布尔逻辑。这样通过复杂的继电器开关之间的相互控制,利用布尔逻辑的数学理论,就可以用硬件来实现人脑的逻辑推理过程,为通用信息处理的工程实现奠定了基础。

在信息处理的硬件技术发展中,综合了数学、化学、材料学、微机械处理和制造、电路与系统设计方法学、计算机科学等交叉学科的半导体集成电路设计与制造技术成为核心力量。用于信息传输和处理的电视、电话网络和计算机网络成为信息时代日常生活的必需品。反过来,计算机与软件技术以及网络通信技术的进步,又促进了半导体集成电路设计技术的发展,从而形成了一个良性循环。在这个良性循环中,一般把这些用于集成电路设计的软件统称为 EDA(electronic design automation,电子设计自动化)工具[24]。随着计算机硬件能力和 EDA 工具的成熟度不断提高,我们可以设计并制造出非常复杂的数字集成电路,从而反过来又促进了计算机硬件和软件技术的发展。Intel 公司的前总裁 Gordon Moore(戈登·摩尔)于 1965 年提出的摩尔定律定量地描述了这种良性循环:单片集成电路上所集成的晶体管的数目随时间指数增长,即每隔 12 至 18 或 20 个月就翻一番。图 1-6 是摩尔先生在 2003 年的 ISSCC(IEEE International Solid-State Circuits Conference,国际固态电路会议)时亲自回顾并预测了他自己提出的定律[25]。图中的曲线显示微处理器和存储器所集成的晶体管数目基本上按指数规律增长,尽管增长速度较之前变缓。

半导体集成电路根据不同的结构和应用可以分为四大类,即微处理器、存储器、逻辑集成电路和模拟集成电路。2006 年全球集成电路的销售额为 2072 亿美元,其中逻辑集成电路的市场规模比例最大,占 28.8%。逻辑集成电路主要包括 FPGA(现场可编程门阵列)、ASIC(application specific integrated circuit,专用集成电路)和各种 ASSP(application specific standard product,专用标准产品)器件[26]。需要说明的是,ASIC 和 ASSP 都属于专用芯片,两者的区别主要在于商业用途:如果一家公司设计的专用芯片不能卖给其竞争对手,那么它就是 ASIC 产品,即为该公司的专用芯片;反之,如果该芯片作为一个标准器件在市场上出售,那么它就是 ASSP 产品,可以卖给多家公司,尽管这些公司之间可能有一定的竞争关系。

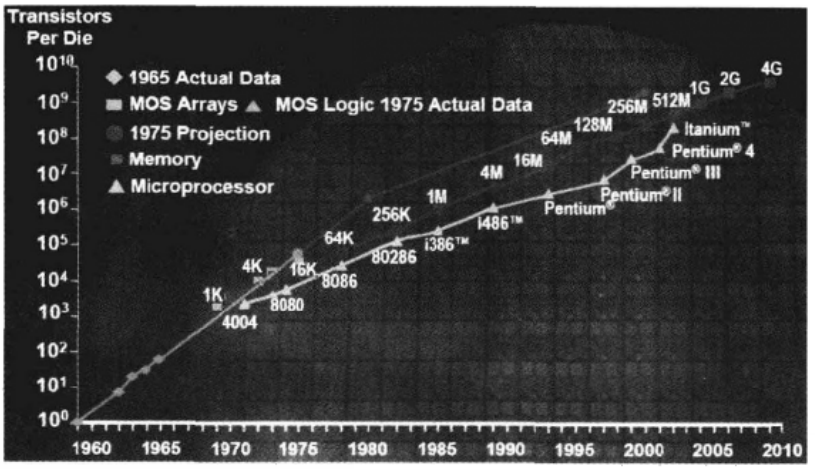


图 1-6 2003 年摩尔在第 50 届 ISSCC 国际会议上对摩尔定律的回顾和预测[25]

根据工业与信息化部的统计信息[27]，我国集成电路产业在“十五”期间进入快速发展阶段。从 2000 年到 2005 年，我国集成电路产业销售收入从 186 亿元提高到 702 亿元，年均增长 30.4%，在世界集成电路产业中的份额从 1.2%提高到 4.5%。市场规模翻了两番，达到 3800 亿元，占到全球的四分之一。芯片设计能力达到 0.18μm(微米)，芯片制造工艺水平达到 12in(英寸)0.13μm，光刻机、离子注入机等关键设备取得重要突破。芯片设计业和制造业比重之和与封装测试业的比重之比从 2000 年的 31∶69 提高到 2005 年的 50.9∶49.1，产业结构更趋合理。涌现出一批具备较强竞争力的集成电路骨干企业，并形成了以长江三角洲和京津地区为中心的产业集聚区。近几年来，我国集成电路市场规模居世界第一，市场增长速度世界第一；但对应的外贸逆差也居我国第一，是石油等制品的 1.7 倍[5]。总体上来说，我国现在正处于发展集成电路产业最好的历史时期。

图 1-7 按照四个不同的时间段，列出了数字集成电路中通用微处理器、专用芯片(包括 ASIC 和 ASSP)、DSP 芯片和 FPGA 及其用户开发环境的发展概况。从图中可以发现以下三个特征。

(1) 随着信息技术的发展，传统的硬件设计和软件开发已经互相渗透。通用微处理器需要强大的编译器支持才能实现各种具体应用程序；专用芯片设计需要各种昂贵的 EDA 工具的支持才能开发出合格的产品；DSP 领域需要 DSP 芯片和 Matlab 等软件工具；FPGA 更是需要器件和编译工具的紧密配合以实现所需要的电路功能。

(2) 随着时间段的变迁，自 20 世纪六七十年代以来，设计层次不断从低往高进展：芯片设计从晶体管级、门级、RTL(register transfer level，寄存传输级)到系统级；软件语言从汇编语言、面向数据和面向对象的 C/C++/脚本语言到 UML(unified modeling language)语言等。FPGA 和 DSP 软硬件系统都能支持更强大的系统级应用设计功能。

(3) 功能不断强大和完善的 FPGA 产品逐渐侵袭专用芯片、DSP 芯片和通用微处理器的市场。由于 FPGA 具有硬件可编程性、并行处理能力强、上市时间(time to market)快等优点，从 2000 年后开始逐渐向另外三大领域扩展。

下面利用图 1-7 简单介绍常见数字集成电路领域的发展概况。它共分为四大块，分别是上边的可编程逻辑芯片及其用户开发环境，下边的通用处理器及其软件开发环境，左边的

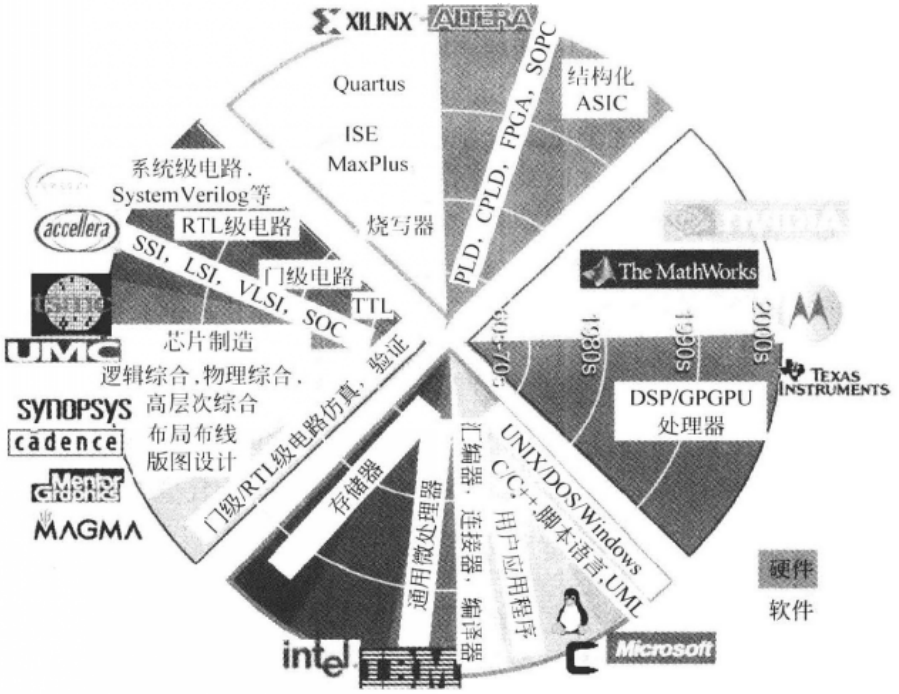


图 1-7　数字集成电路软硬件的发展概况

专用芯片和 EDA 领域和右边的 DSP 芯片及用户开发环境。

(1) 通用微处理器和软件领域：自从 Intel 于 1971 年 11 月推出第一款微处理器 4004 以来，通用微处理器的规模基本上符合摩尔定律指数增长，如图 1-6 所示。对于存储程序计算机模型来说，程序本身和用户数据都需要存储器的支持，因此存储器的规模也是指数增长，如图 1-6 所示。微处理器和存储器构成了存储程序计算机的核心硬件结构。在软件开发方面，“二战”刚结束时计算机主要是通过手动硬件开关或者打孔纸带进行编程，编程语言仍以机器语言或汇编语言为主，程序代码经过汇编器(assembler)和连接器(linker)产生最终的可执行程序，因此编程效率很低。1954 年，IBM 的工程师们在 IBM704 计算机上设计了世界上第一个高级语言 Fortran 及其编译器。后来出现的 C/C++、Java、UML 以及各种脚本语言如 Shell、Tcl(tool command language)、Perl(practical extraction and report language)和配套的编译器、解释器和集成开发环境(integrated development environment，IDE)等，均大大提高了软件编程和调试效率。另外，以 Windows/UNIX/Linux 为代表的操作系统为普通用户使用计算机提供了方便，这样计算机能够轻易地进入实验室、办公室和家庭。1990 年开始，基于通用计算机和软件的信息处理平台在全球范围内迅速普及。除了 Linux 和微软的图标，图 1-7 右下角还标记了在 Windows 操作系统上提供 UNIX 功能的 Cygwin 工具图标。

(2) 专用芯片设计和 EDA 领域：从芯片设计和制造的角度来看，通用微处理器的开发离不开芯片技术。或者也可以说，通用微处理器的发展也会带动专用芯片的设计和制造水平。专用芯片的规模从小规模(small scale integrated circuit，SSI)、大规模(large scale integrated circuit，LSI)，发展至超大规模(very large scale integrated circuit，VLSI)、SoC

(system-on-a-chip)和 NoC(network-on-a-chip)。对于 SSI 或 LSI 电路,原理图和版图还可以用全定制的方法完成设计。例如 Intel 的 4004 芯片包含 2300 个晶体管,电路和版图都是全定制设计的。为了满足芯片规模不断扩大的需求,电路设计也从最初的晶体管级,上升到门级,再到 RTL 级。RTL 级描述的电路行为经逻辑综合 EDA 工具用不同的逻辑门电路搭建起来。相应地,芯片仿真与验证工具从早期晶体管级的 Spice 仿真、门级电路上升到 RTL 级电路功能仿真和时序分析验证。RTL 级电路描述需要经过前端的逻辑综合工具和后端的布局布线、时序分析和版图设计才输出 GDSII(graphic design system II)格式的版图文件。芯片制造商就可以根据这个 GDSII 文件生产出需要的集成电路。目前也有一些 EDA 工具直接支持系统级语言描述,例如 SystemVerilog、SystemC、ANSI 标准 C 语言等。这些能够直接从高层系统级语言描述自动产生 RTL 级硬件电路描述的软件就称做高层次综合(high-level synthesis)工具。它们相对于传统的逻辑综合工具来说把输入语言的层次从逻辑级提升到了高层次的系统级。语言描述层次越高,电路的设计规模就可以增大,但是电路的性能却比晶体管级的设计电路要低很多。这就好像 C 语言写的程序要比直接用汇编语言写的程序效率低一样。但是现实的情况是世界上使用诸如 C 语言的用户远远要大于汇编语言的使用人数。因此,如何直接支持系统级语言描述,进行高层次综合输出高效率的硬件电路描述是 EDA 公司所要解决的难题。目前全球前四大 EDA 公司分别为 Synopsys、Cadence、Mentor 和 1997 年成立的后起之秀 Magma。早期的集成电路公司都拥有自己的设计和制造设备。但是为了进一步提高管理效率和专业程度,80 年代后出现了一些基于代加工(fabless)模式的芯片设计公司,这些公司不从事芯片的制造。这种芯片设计和芯片制造的分工管理方式对提高产品的效率、降低芯片的成本可以起到很好的作用。根据市场研究公司 Gartner 的统计,2007 年全球前四大芯片制造商为中国台湾积体电路制造股份有限公司(TSMC,简称台积电)、中国台湾联华电子公司(UMC,简称台联电)、中芯国际集成电路制造公司(SMIC)和新加坡的特许半导体制造有限公司(Chartered Semiconductor)。

(3) DSP/GPGPU 处理器及其 Matlab 和 CUDA 开发环境领域:随着通用微处理器和专用芯片的设计水平不断提高,两者之间性能和灵活性的差距越来越大,如图 1-8 所示。一方面,通用微处理器具有广泛的灵活性,但是计算效率较低。通用处理器内部的大部分晶体管都不再直接参与计算,而是为 ALU 算术逻辑单元准备数据。也就是说,通用处理器内部的大部分晶体管面积都用于高速缓冲器(cache)、存储器接口控制逻辑、分支预测逻辑、指令译码逻辑等功能,只有很少的一部分晶体管面积直接用于计算,因此通用处理器的计算效率就较低。另一方面,专用芯片的信息处理效率很高,但是不够灵活。每一种应用,哪怕只有细微的差别,都需要重新设计制造不同的芯片。对于 DSP 应用领域,它对数字信号处理效

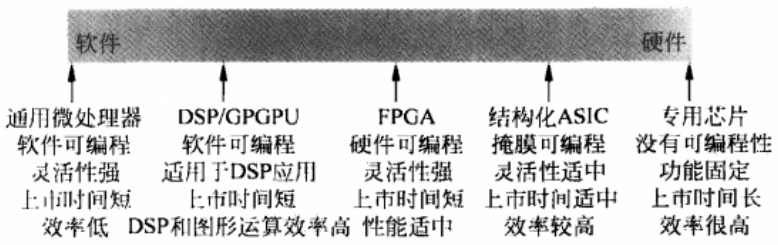


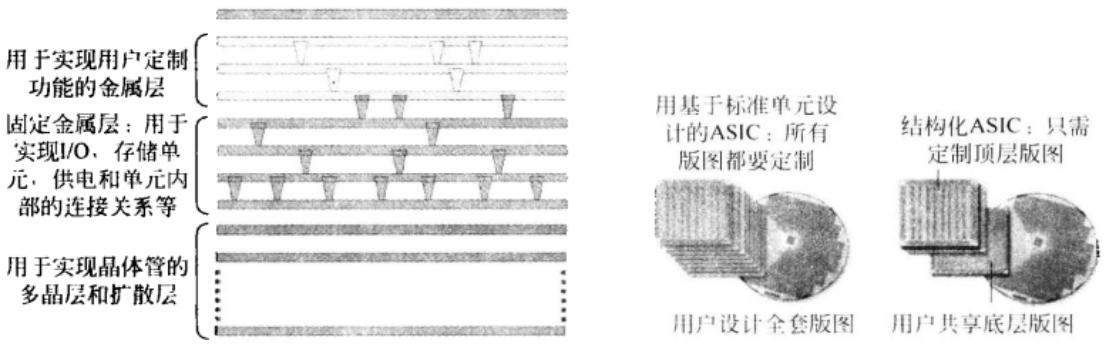
图 1-8 各种信息处理平台的性能比较

率有很高的要求，同时又需要有一定的灵活性，能够支持任意的 DSP 算法。因此 20 世纪 80 年代，以 TI 公司为代表推出的 DSP 芯片便应运而生。和通用处理器的结构相比，DSP 处理器一般采用独立的数据总线和程序总线，即哈佛（Havard）结构，可以同时进行指令和数据的读取操作，提高了指令执行速度和数据计算能力。新型 DSP 结构还具有 SIMD（single instruction multiple data，单指令多数据）的功能，即由单一指令部件同时控制多个处理单元，从而让同一指令能够操作多个数据。另外 DSP 处理器一般具有多个专用的乘加器，能够在一个时钟周期内完成乘加操作，特别适合于滤波器、矩阵运算等场合。在数据格式方面，新型 DSP 芯片能够直接支持浮点数操作，进一步提高浮点数计算的性能。对通用微处理器来说，浮点数运算需要借助于内嵌的浮点数协处理器来实现。在软件工具方面，不同的 DSP 公司都各自提供用户开发环境，支持从高级语言到汇编语言的编译和调试功能，产生的可执行程序可以直接在开发板上运行调试。这些 DSP 开发板一般都已经提供 A/D 和 D/A 转换模块，可以直接输入输出模拟信号。从 DSP 系统前端设计来说，Matlab 公司提供的各种软件包提供了跨 DSP 平台的快速系统设计功能。Matlab 提供的各种软件包是目前全球最好的数据运算和处理工具，特别适合于 DSP 应用和大量数据计算的场合。近几年，由于对图形信号处理的效率需求越来越高，常用的 DSP 处理器无法满足计算效率和灵活性的需求。因此 GPGPU 处理器能够有效地解决并行计算效率和灵活性之间的矛盾，成为目前非常热门的多核处理器计算平台。

（4）可编程逻辑及其用户开发环境领域：随着专用集成电路的复杂度、性能、设计成本、上市时间以及对 EDA 工具的综合要求不断提高，专用集成电路的缺点越来越明显。如果用户需要修改芯片的部分功能，那么整个芯片都要重新设计和制造。另一方面，不管是通用微处理器或者是 DSP 处理器，它们所处理的数据都是粗粒度，即基于 8 位、16 位或 32 位的数据格式，它们并不支持大量控制电路中的细粒度需求——对每一位信号的状态都能控制。因此，20 世纪 90 年代以来，作为硬件细粒度可编程的 FPGA 器件成为专用芯片设计的原型验证平台。FPGA 提供的现场可编程性是指用户在现场使用芯片的时候，在不改变芯片硬件运行环境的条件下，就可以通过硬件提供的可编程性来实现不同功能的属性。硬件工程师完成 RTL 代码后，首先利用 FPGA 的现场可编程性进行功能验证。这种基于 FPGA 的原型验证方法要比 RTL 仿真工具，如 Mentor 公司的 ModelSim 仿真速度快很多。另外，ModelSim 仿真工具通过的 RTL 代码并不能保证是可综合的，而通过 FPGA 编译流程的代码则能够保证是可综合的。再且，随着深亚微米集成电路设计成本的急剧上升，FPGA 器件开始从硬件原型验证往系统级应用扩展，并且开始应用于一些终端产品，例如汽车电子、消费类产品等。在硬件结构方面，FPGA 器件一方面嵌入了存储器和微处理器，例如 Altera 公司的 Excalibur 和 Stratix 系列产品拥有硬 ARM 核或 Nios Ⅱ 软核，Xilinx 公司的 Virtex 系列产品拥有 PowerPC 硬核或 MicroBlaze 软核；另一方面又嵌入了多个 DSP 领域专用的乘加器单元，大大提高了对粗粒度数据的处理能力和效率。这些 DSP 模块可以根据不同的需求配置为不同的数据宽度，满足不同应用的需求。在 FPGA 的用户开发环境方面，Altera 和 Xilinx 公司都推出支持系统设计的软件工具，这些工具不但支持传统 RTL 的编译流程，具有完善的可重用 IP（intellectual property）库，而且还支持 C/C++语言的编译和调试，以及和 Matlab 工具的接口，极大地开拓了 FPGA 工具在 DSP 领域的应用范围。

除了以上四种数字集成电路计算实现方式外，随着半导体工艺水平的不断进步以及设

计和制造成本的急剧上升，FPGA 和专用芯片之间的差距也明显增大。也就是说，在 FPGA 和专用芯片之间，我们还希望找到另一种实现方式：它的可编程性介于 FPGA 的现场可编程性以及专用芯片的不可编程性之间，即具有一定的可编程性和灵活性；上市时间可以比 FPGA 要长，但是性能要比 FPGA 好。这种实现方式就是“结构化 ASIC(Structured ASIC)芯片模式”，如图 1-8 所示。尽管目前它的市场份额还非常地小，或者说在商业上目前不很成功，但是它作为一种商业产品的基本思想还是有一定的借鉴之处。当 FPGA 和专用芯片之间的差距进一步扩大时，结构化 ASIC 就开始拥有一定的生存空间。对于结构化 ASIC 实现方式来说，它的可编程是通过掩膜来实现的，芯片在设计时可以灵活改动，但是芯片制造完成后就不再具有可编程性，因此电路性能要比 FPGA 好。但是和专用芯片相比，结构化 ASIC 版图中的底层电路是事先设计好的。用户电路只能共享这些底层电路。不同的电路功能是通过上层的数层掩膜来实现。因此从设计和制造的时间和成本来说，它要比所有版图都要专门设计和制造的专用芯片低很多。Altera 公司的网站提供了专用芯片的标准单元设计方法和结构化 ASIC 实现途径的比较，如图 1-9 所示。对于标准单元来说，所有的掩膜版都是用户自定义的，具有最高的面积、时序和功耗性能，但是上市时间长、成本高。结构化 ASIC 只允许用户修改上面数层掩膜，底层的一些掩膜，包含 Poly(多晶)和 Diffusion(扩散)层都是事先定制好的，且经过了测试验证。因此结构化 ASIC 的优点是上市时间短，掩膜数量的减少使得成本比标准单元设计低，芯片的性能比标准单元芯片要差。但是由于不具有现场可编程性所需要的可编程开关，因此结构化 ASIC 的时序和功耗等性能比 FPGA 要好。如 Altera 的 HardCopy 产品就属于结构化 ASIC 系列。它可以根据用户在 FPGA 上调试成功的电路，去除了 FPGA 中的可编程性，从而实现了功能相同但性能改善的专用芯片，节约了大量的纳米级专用芯片开发成本。



(a) 结构化ASIC的基本结构　　(b) 两种AISC所需要掩膜版数量的比较

图 1-9　基于标准单元的 ASIC 设计方法和结构化 ASIC 的比较(www.altera.com)

1.1.4　集成电路的现场可编程性需求

到 20 世纪 70 年代末或者 80 年代初，数字电路大部分是由分立元件构成。从 20 世纪 80 年代开始，由于计算机及其软件技术的发展，数字电路本身的设计也可以利用计算机来辅助完成，从而就诞生了 EDA 这个产业，并且促成了专用集成电路的设计和发展。利用 EDA 软件，用户可以根据不同的需求，设计专用集成电路。相比于由分立元件组成的数字电路，专用集成电路的可靠性、性能、功耗等均有明显的优势。从 80 年代开始，各种 ASIC

和 ASSP 产品迅速替代了原来由分立器件设计的数字电路。并且在摩尔定律的驱动下，数字电路的集成度、性能和成本都不断优化，从而形成了专用集成电路的黄金时代。到了 90 年代，基于分立元件设计的数字电路基本上没有了，一般的数字系统主要由微处理器或微控制器、存储器和各种 AISC 器件组成。

半导体集成电路的设计过程是非常复杂的，并且我们还不能保证所设计的芯片一定是最优的。因此，对于这一类计算复杂度很高的集成电路设计问题，一般采用分而治之(divide and conquer)的策略。也就是说，把这个复杂的问题划分为一些子问题，然后分别用不同的办法来解决这些子问题，可能有些子问题还需要进一步划分。基于图 1-7 的理解，集成电路的设计从系统应用需求到应用产品可以按照图 1-10 进行划分。

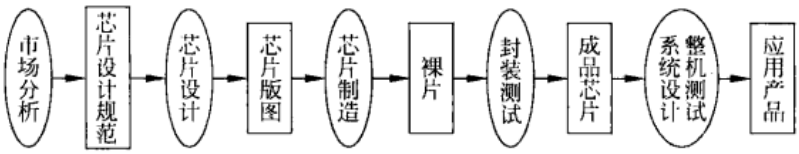


图 1-10　芯片的设计及其生命周期

首先根据市场需求确定系统级行为，提出设计规范。此时系统行为一般采用系统级描述语言，例如利用 C/C++、Matlab 工具进行仿真，明确了系统的应用环境及其输入输出功能。在这个阶段，还没有引入系统所需要的同步时钟。

第二步是根据系统设计规范利用硬件描述语言手工编写 RTL 级代码。目前主流的硬件描述语言是 Verilog 和 VHDL。在 RTL 描述中，一般电路都引入了同步时钟，确定了每个时钟周期各信号的行为，例如可以用 for、while 等循环语句来描述行为。除了代码编写的实现方式外，这一步也可以采用原理图(schematic)的方式画出整体或者部分模块的电路图，或者重用一些 IP 核。

第三步是由 RTL 描述到门级电路的实现。这就是说 RTL 所描述的电路功能全部要用硬件单元库(cell library)所提供的硬件电路单元来实现。这些硬件单元可能包括各种类型的宏单元或者 IP 核等。例如 RTL 描述中的各种 for、while 循环语句都被展开为硬件电路实现。如果 RTL 描述中用到一些存储单元，那么在硬件上就直接利用合适的寄存器或存储器来实现。一般来说，由 RTL 描述到门级电路的实现由逻辑综合工具自动完成。用户可以利用这个工具设置约束和各种参数，从而优化电路的整体性能。

第四步是门级电路到晶体管电路的实现，并进行布局布线。对于小规模的晶体管级描述电路，传统上可以用 Spice 工具进行更为精确的仿真和验证。例如在建立单元库时就需要 Spice 仿真结果来提高库参数的精度。逻辑综合后的门级电路可以根据单元库所提供的信息及其参数设置值直接转为晶体管实现方式，并且根据这些单元的尺寸大小和连接关系进行合理的布局布线。一般这一步是由后端的布局布线工具自动完成。布局布线后的电路可以由仿真工具进行后端的时序仿真，并可以运行静态时序分析工具以确定电路关键路径和最高工作频率。对于深亚微米电路来说，一般还要运行功耗分析工具，用于衡量电路的静态和动态功耗，从而优化芯片的功耗性能和可靠性。

第五步是从晶体管电路到版图文件输出。此时电路就被转化为各种多边形和过孔等单元组成的版图描述形式。这一步骤一般由版图工具自动完成。对于全定制设计的电路，一

般还需要手动画版图并进行编辑，从而优化电路的性能，并尽量缩小版图面积。GDSII 是版图描述常用的标准格式。从芯片设计的角度来说，它就可以把优化后的 GDSII 文件递交给芯片制造商进行后端的制造工艺流程。

第六步是从版图描述到掩膜制造。这一步主要是芯片制造商根据版图文件制作掩膜的过程。一般来说一个芯片需要数十层掩膜版。掩膜版类似于传统照片的底片功能，利用激光(波长一般为 193nm)影像技术、离子注入、腐蚀与扩散技术、化学机械抛光技术(chemical-mechanical polishing，CMP)、光学近似修正技术(optical proximity correction，OPC)和相移掩膜技术(phase shift mask，PSM)等其他复杂的光刻技术在单晶硅半导体晶圆(wafer，或称做晶片)上制作所需要的晶体管及其层次化的金属互连线。芯片制造商对制造后的晶圆进行测试从而标记存在制造缺陷的裸片。由于一个晶圆不可能刚好容纳所有完整的矩形裸片阵列，因此晶圆边上一般都存在着不完整的裸片。

第七步就是把矩形裸片(die)从晶圆上切割下来并封装成芯片。一片晶圆一般可以制作上百个裸片，不过裸片还需要经过封装后才成为一个可用的芯片。裸片的四周一般留有引脚(pad)，这些引脚在封装时通过铝线或金线连接到外面的管脚，如图 1-11 所示。封装后的芯片还要经过自动测试设备(automatic test equipment，ATE)进行功能和性能的测试和定标。由于芯片资源的规模随着芯片的面积增长，而面积是边长的平方关系，其增长速度要快于芯片周长的增长速度。因此，20 世纪 90 年代后，芯片集成度的不断增加促使芯片封装采用 BGA(ball grid array)技术，即球栅阵列封装，从而使得芯片的引脚数目能够和芯片的资源同步增长。

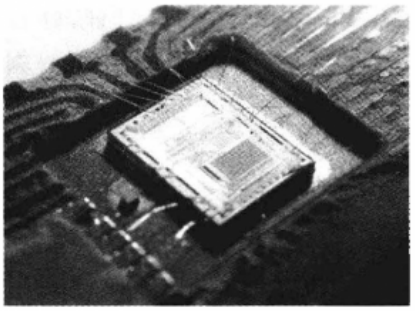


图 1-11 Intel 公司 8 位控制器剖盖后的裸片和封装
(http://en.wikipedia.org/wiki/Integrated_circuit)

最后一步是芯片的测试与定标。由于芯片制造时的工艺偏差和硬件单元库数据存在的误差，芯片在销售前需要利用自动测试设备检测功能和性能。功能正确但是性能不同的芯片就标记为不同的速度等级。系统设计工程师就利用这些芯片和其他元件设计具体应用的印刷电路板(printed circuit board，PCB)，进而完成整机的制作、调试、测试和销售。

如上所述，集成电路的生命周期一般要经过设计、制造、封装测试和销售应用等过程。然后公司再根据产品的市场销售情况和用户反馈意见开始下一款芯片的设计。对于成熟的公司来说，产品的市场分析、研究开发和量产过程都是同步进行的。公司所提供的系列化产品能够满足不同用户的需求，从而使得产品更具竞争力。当然，芯片的销售价格和利润与芯片的性能、市场竞争力和售后服务等密切相关。这里需要强调的是“上市时间”这个非常关键的因素[28]。如图 1-12 所示，可采用简化的产品销售利润模型——线性模型来计算一种芯片的利润和上市时间的关系。图中假设产品及时上市的情况下在 T 时间后达到最高利润点 T，然后再经过 T 时间后被新一代产品所淘汰而退出市场。如果该产品延时 D 时间才上市，那么同样假设经过 $(T-D)$ 时间后达到最高利润点 $(T-D)$。这样该产品在整个生存周期内的利润就等价于三角形的面积。两个三角形的面积分别为

$$R_{\text{及时上市}} = \frac{1}{2}(2T \times T) = T^2 \tag{1-13}$$

$$R_{\text{延时上市}} = \frac{1}{2}(T + T - D) \times (T - D)$$

$$= \frac{1}{2}(2T^2 - 3TD + D^2) \tag{1-14}$$

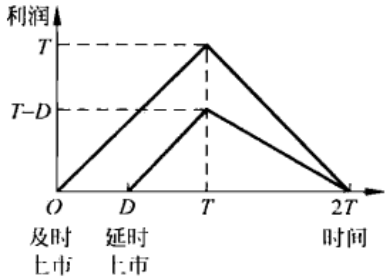


图 1-12　上市时间对产品利润的影响

将以上两式相减，就得到延时上市引起的利润损失比例为

$$L_{\text{利润}} = (R_{\text{及时上市}} - R_{\text{延时上市}})/R_{\text{及时上市}}$$

$$= \frac{1}{2T^2}(3TD - D^2) \tag{1-15}$$

假设 $T=26$ 周，$D=4$ 周，由式(1-15)可知利润损失约为 21%。如果 $D=10$ 周，即一个产品大约在整个生命周期的近 20%时才进入市场，那么利润损失为 50%，即失去了一半的利润。由此可见上市时间对于产品的利润是多么的重要。

随着半导体工艺技术的更新，芯片的集成度和处理能力都不断提高。但是出于市场因素的考虑，这并不意味着最新工艺、最高性能的产品一定具有最强的竞争力。一个成功的产品一定要根据市场需求的规模来确定合适的上市时间、投资力度及其相应的设计进度。如前所述，信息技术最基本的要求是如何在硬件上实现“0”和“1”两种开关状态。香农最开始是采用当时普遍使用的继电器开关。如今集成电路普遍采用的是 MOS 晶体管开关，就是用一个可控的沟道来实现“开”和“关”两个状态。相比机械接触式的继电器开关，晶体管开关具有更高的速度、可靠性和寿命。但是晶体管开关只有当它的性能和可靠性都达到了用户可接受的标准后才被普遍采用。集成电路的设计和制造也经历了漫长的技术改进，只有当它的成本和可靠性都具有一定竞争力的时候才能代替原有的基于分立元件设计的逻辑电路。又如早期电脑的机电式鼠标是通过圆球的机械运动来移动光标位置，那时光电式鼠标还比较昂贵。现在的光电式鼠标均由发光二极管、透镜等光学器件和控制芯片组成，它每秒能够采样分析数千次图像，控制芯片进行图像比较后判断鼠标是否移动和移动的方向。因此它的性能和可靠性明显优越于机电式鼠标。但是只有当它的成本能降到和机电式鼠标相差不多的时候，才能赢得市场优势。现在光电式鼠标基本上完全取代了传统的机电式鼠标。

为了能够具备更强大的信息处理能力，在芯片内尽可能放置更多的晶体管开关。但是晶体管尺寸受到半导体制造工艺技术的限制。在芯片制造时，MOS 晶体管的沟道长度就可以表征芯片的集成度，通常把它称做特征尺寸。国际上半导体工艺的特征尺寸从 1971 年第一款通用微处理器 4004 所采用的 10μm 工艺，发展到 2007 年的 65nm 工艺水平，特征尺寸基本上是随时间指数减少[25]。对于特征尺寸的不同定义，可以参照国际半导体技术规划(international technology roadmap for semiconductors, ITRS) 2001 年版本的总体报告[29]。

为了对特征尺寸有感性认识，可以参考我们所熟悉的一些尺寸。例如头发的尺寸大约为 100μm，我们肉眼能看见的最小尺寸大概是 50μm，红细胞大小约为 7μm[25]，红色可见光的波长约为 0.6μm，目前集成电路制作掩膜的光刻激光波长为 0.193μm 或 193nm。在半导体工艺特征尺寸达到 0.13μm 以前，每一次的工艺更新都会减少单个晶体管的版图面积，相应的负载电容也会减少、晶体管的开关速度加快并且控制单个管子状态的功耗也下降，使得

芯片的集成度和性能均得到提高。但是在 0.13μm 工艺以下，由于漏电流和功耗以及可制造性设计(design for manufacturability，DFM)等问题，专用集成电路设计和开发面临以下瓶颈问题。

(1) 设计成本成倍上升：当半导体工艺技术进入超深亚微米或者纳米尺度，芯片设计和制造成本急剧上升。2006 年 65nm 的芯片设计和制造成本，包括购买 EDA 软件工具、设计验证、流片、产品测试、费用合计约 3500 万美元[30]，如图 1-13 所示。

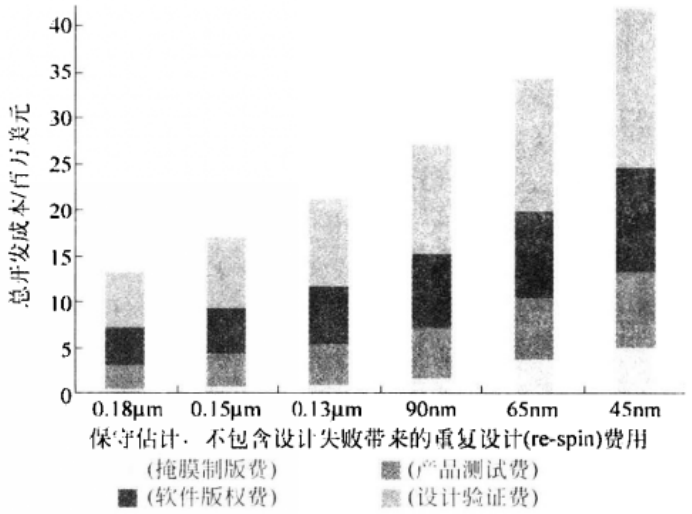


图 1-13 美国 *COTS* 杂志于 2006 年对集成电路研发所需要的成本估计[30]

(2) 芯片生命周期缩短：对于日益重要的消费类电子芯片，例如手机芯片，其生命周期一般持续不到 12 个月。

(3) 开发时间不断加长：由于芯片尺度越来越细，芯片设计验证和制造变得十分复杂，一般至少需要 1 年的时间，以保证芯片功能的正确性、可制造性和可靠性。

以上瓶颈问题要求超深亚微米或纳米级集成电路必须具有大批量的市场需求才能产生足够的经济效益，例如微处理器、存储器、电视或游戏芯片、手机芯片等。对于大量中小批量需求的应用芯片，用户希望买到芯片后能够根据当时的需求和具体应用可以灵活地调整芯片的功能、时序等性能指标，在不同的应用之间能够充分地共享芯片提供的各种硬件资源，即芯片需要具有现场可编程性。对于专用芯片来说，每一次功能修改——哪怕是很小的改动，都需要重新设计、流片、测试等步骤。这个修改过程在图 1-13 中叫做 re-spin。如果芯片能够具有现场可编程性，那么用户可以根据当时的市场情况，灵活地调整芯片的性能，提高产品的竞争力。2007 年 12 月，国际半导体技术规划报告基于芯片设计中的可重用性和可靠性需求，指出未来 SoC 芯片中将嵌入更多的可编程逻辑资源。该报告预计到 2015 年，48%的芯片功能是采用可编程逻辑(reconfigurable logic)来实现的。

1.2 存储器和现场可编程性

“可编程性”(programmability)或者“编程”(programming)的内涵很广，它们的基本意思是在给定的环境和条件下实现特定功能的创作过程，或者是利用给定的资源实现用户具

体目标的定制过程。“用户定制”(customization)是这个过程的最基本性质。它表示一项任务的完成不是从零开始,而是利用已提供的各种条件进行配置或定制的过程。在日常生活中,我们经常不自觉地得益于这种可编程性。如果我要打开房间里的电灯,只需要拨动电灯的开关就可。开关拨通后,电灯就会亮;开关关掉后,电灯就会熄灭。对于用户来说,他不关心电灯的交流电源是如何供电的,如何采取过流保护,电线又是如何走线的,电灯的功率是多少,开关的内部结构如何等,他只要操作所需要的开关就可。因此用户拨动开关的过程就是在已经把电灯安装好的条件下如何利用开关操作电灯的过程。这就是一种“编程”过程。电灯的供电方式、过载过流保护方式、线路布线路径、电灯的功率和开关内部结构等都是安装电灯时所完成的——这些是提供给用户“编程”用的环境。一般用户在“编程”时是不会去改变这些环境的。相对于拨动开关的“编程”过程来说,电灯所需要的线路系统的安装过程就是一种“实现”。从用户的需求来说,“编程”操作往往要比“实现”过程简单很多。当然,这种“编程”操作也可以很复杂。相对于简单的“开灯”操作,舞台上的灯光控制就算是一种专业技术了,此时对各种舞台灯的线路安装需求也就更高了。对于普通电灯的用户来说,通过电灯开关所提供的可编程性,我们只需要知道如何操作开关的状态就可。这就大大提高了使用过程的便利性和灵活性。又如,手机表面更是复杂的按钮编程系统。用户只需要触摸手机表面的各种按键,就可以进行各种操作,而不需要知道手机的内部结构。因此手机的功能使用和表面操作的设计非常相关。苹果公司的 iPod 产品曾经非常畅销,主要是由于它操作简单,且具有很友好的用户界面。当然,用户的“编程”操作和提供编程环境的“实现”过程的关系是相对的。例如对于日常的照明用电来说,用户拨动开关的操作是一种“编程”过程。而相应电路的线路安装是一种“实现”过程。但是对于线路安装人员来说,他不需要理解为什么电灯通电后会发光的原理,他只要把电灯安装在插座上即可。电灯通电后会发光这个过程是电灯设计人员和制造商的工作。此时,电灯的安装是一种“编程”过程——只要把所需要的电灯安装到合适的插座上就可以了。而电灯的设计和制造就是一种“实现”过程。这个过程远比电灯的安装要复杂很多。

另外从英文 program 这个词似乎更容易理解“编程”和“实现”的关系。如果所提供给用户定制的对象是我们日常中的生活环境,那么电视、电影或者广播节目制作人可以编制各种节目。此时中文把 program 翻译为“节目”。如果给定计算机及其运行软件资源,那么软件设计人员就可以为了实现特定的任务编写程序。中文把这个实现过程称为“编程”。编程者成为 programmer;他编程的成果就是“程序”(program)。编程的时候,用户可以不关心他所用的高级语言如何转为处理器的指令,这些指令如何在处理器上运行,微处理器内部的晶体管是如何进行开关操作的。编程时用户的主要精力集中于软件的算法实现。这种编程的过程和一个导演安排演员对日常生活中的人物和事件进行节目编制,舞台灯光师根据剧情来调整灯光的亮度和角度是很类似的。就如 Donald Knuth 所言:“计算机编程是一种艺术形式,如同创作诗歌和音乐。”

对于给定 FPGA 器件所包含的可编程资源,用户可以利用所提供的软件工具进行编程,从而实现具体的电路功能。这种电路就称为可编程电路。相对于软件人员编写代码的过程来说,我们可以把这种通用型电路资源理解为“硬件可编程”电路。从“可编程性”的角度来看,硬件可编程性必须满足以下条件。

(1) 可编程性的实现:这是要解决如何在硬件上向用户提供可编程性的问题。和软件

可编程性一样，目前最常用的硬件可编程性的实现方式还是基于存储器。用户只要写入不同的可编程数据，就可以配置为不同的电路实现。实现可编程性的存储单元有多种结构，例如 Fuse/Antifuse（熔丝/反熔丝）、EPROM（erasable programmable read-only memory）、EEPROM（electrically erasable programmable read-only memory）、Flash、SRAM 等。

（2）可编程硬件结构：这是可编程配置数据所控制的对象。对于软件可编程性来说，可编程数据主要是通过指令流的方式控制微处理器的运行。对于硬件可编程来说，主要包括可编程芯片的硬件架构，例如可编程逻辑结构、可编程互连结构和可编程输入输出单元结构等。这些可编程资源经过可编程数据的配置后就实现了特定的逻辑函数，连接方式和输入输出功能。

（3）可编程数据自动产生工具：对于电路设计者来说，需要提供一套软件工具及其用户开发环境。它可以根据用户的具体任务需求，自动产生可编程数据。对于软件可编程性来说，产生二进制可执行程序的这套工具就叫做软件编译器。对于硬件可编程性来说，这就是可编程器件的 EDA 工具，例如 Xilinx 公司的 ISE 和 Altera 公司的 Quartus 工具等。

本节从用于可编程性实现的基本存储器结构开始介绍。下一节介绍面向软件可编程性的通用微处理器编程和 DSP 设计技术，包括基本可编程硬件结构和软件开发工具。最后介绍与可编程电路设计相关的专用集成电路设计技术和系统级 FPGA 计算平台的特点。

从信息处理的灵活性和现场可编程性的需求来说，不管是 DSP、GPGPU 还是 CPU 或者其他可编程器件，我们首先要实现这样一种芯片结构：在不改变硬件本身的前提下能够方便地实现不同的“0 / 1”信息处理方式。也就是说，能够在固定的硬件结构上灵活地实现信息的各种处理方式。这样，对于同一个芯片来说，只要写入不同的编程数据，就相当于实现不同的信息处理功能。

对于半导体集成电路来说，存储器是最基本的可编程器件。我们把它所存储的信息称为数据。从信息的存储效率、速度、便利性来说，存储器技术比活字印刷术要先进很多。对于活字印刷，虽然它比之前的文字存储方式的效率高很多，但它是基于固定的硬件单元——铅字。一个单元只能表达一个固定的文字。而对于存储器来说，一个硬件单元既可以存“0”信息，又可以存“1”信息。这个过程不需要对硬件结构作任何的改动。这就像普通纸张和电脑的显示屏一样。一张白纸写了东西后就不能在上面再写其他东西，除非预先擦除。但是电脑的显示屏能动态显示图形信息。只要通过存储器提供一定序列的“0”和“1”信号，再经过一系列的数据处理过程，就能够在显示屏上显示各种图形，不管是静止照片还是动态图像。这些例子都说明了存储器技术给我们带来的可编程性和便利性。

从学术研究的角度来看，一个静态事物或一套理论的效率往往取决于它所能应用的范围大小。例如英国天才科学家牛顿的伟大之处，就在于他提出的万有引力理论，用类似于 $F=ma$ 这样一个非常简单的数学表达式就能够准确计算大至天体、小至灰尘颗粒的运动方式。如果一个事物或环境，或者是一种理论能够用越简单的方式，描述越复杂事物的运动变化方式，那么就越能体现出它的优越性。在物理学领域，另一位天才科学家爱因斯坦在生命的最后 30 年置身于研究统一场论——他试图用一种理论来统一牛顿的万有引力理论和麦克斯韦的电磁理论。虽然他没有成功，但是对统一场论孜孜不倦的追求一直是他的梦想。对于统一场论以至万有理论的追求，即寻找一种理论就能够描述自然世界的四种力——万有引力、电磁力、强力和弱力，也是另一位英国科学家霍金的梦想。但是在 2002 年 8 月 17

日北京国际弦理论会议上，霍金指出在物理学领域可能存在与哥德尔不完全定理类似的规律，因此不太可能建立一个描述宇宙的大统一理论[31]。从信息处理的角度来看，现场可编程性也就是想达到这样一种目的：提供一种固定的硬件资源和用户编程环境，尽可能满足用户的信息处理需求，在不需要改变硬件环境的情况下，能够实现尽可能多的应用，即应用范围越广越好。对于CPU和软件可编程性来说，它是目前最通用的一种计算实现方式，即它能够实现任意可用软件语言描述的计算。但是它的计算效能已经无法满足日益增长的信息处理需求。虽然后来出现了多核DSP或者GPU处理器，但是它们的应用范围主要局限于数字信号处理系统，还没有达到类似于CPU的通用型计算实现方式。GPGPU就是要利用GPU处理器实现通用型计算的目的。另一方面，FPGA提供了通用的硬件可编程性，能够实现比CPU更高的计算效率。但是FPGA的便利性远没有达到CPU和软件可编程性这样的成熟度。因此，如何提出一种更为通用、高效的信息处理硬件结构及其用户编程环境，是未来实现数字集成电路可编程性和灵活性的长远目标。

不管是软件可编程性还是硬件可编程性，目前都是通过存储器的方式来实现的，例如FPGA中用于配置的SRAM阵列，微处理器中基于SRAM的高速缓冲器以及基于DRAM(dynamic random access memory)的主内存(main memory)。这些存储单元阵列的用途和访问方式各不相同。FPGA中的SRAM配置单元阵列是用于存储硬件可编程信息。为了提高编程下载速度，它是以“帧”(frame)作为最小读写单元，一般一帧包含数百至数千位配置信息。SRAM配置单元的结构及其操作见2.2.2节。通用微处理器的主内存主要用于存放可执行程序和用户的数据，读写操作一般是以“字节”作为基本单位。但是DRAM单元为了避免电容放电而造成数据丢失，需要动态刷新操作，因此它的访问速度要远远落后于通用微处理器的工作速度。为了弥补通用微处理器和主内存的速度差距，20世纪90年代开始通用处理器就利用运行数据的局域性(locality)在片内或片外采用了层次化的多级高速缓冲器(cache)。所谓数据局域性就是在一定的运算时间内，处理器不可能需要访问所有的用户数据。也就是说，处理器在一段时间内访问的数据只是所有用户数据中很少的一部分。高速缓冲器访问时一次不只读写一个字节，而是以“缓存行”(cache line)的格式进行访问。一个缓存行一般包含数十至数百字节。

常用的存储器基本接口如图1-14所示。它一般包含基本的地址(address[])、输入数据(data[])、输出数据(q[])、读写控制信号(wren)和同步时钟信号(clk)，其中输入和输出都是以字节为单位的数据，地址线的大小取决于存储器的阵列规模。当写使能信号有效时，地址线上的值就作为写入数据的存储器地址。当写使能信号无效时，输出数据就是地址线所对应的存储值。为了提高工作速度，DDR(doube-date rate)存储器可以在一个时钟周期内读写两次数据。另外还有更为复杂的双口电路，使得存储单元的数据可以同时进行读写操作。当然读写操作不能同时对相同的地址单元进行。

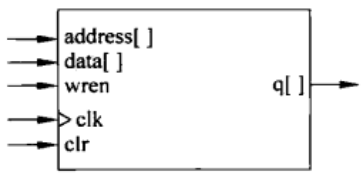


图1-14　存储器的基本访问接口

根据存储器的数据宽度和阵列大小，我们可以得到图1-15所示的层次结构。为便于定性理解各存储层次的相对性能，图中标注的数据，包括数据访问方式、阵列规模和访问时间大致为2008年前的参考数据。层次结构中的最上层是SRAM单元。标准SRAM单元的核心结构是由两个首尾相连的反相器组成，如图1-16所示。图中每个反相器由两个晶体管

实现，而 M5 和 M6 两个晶体管用于控制数据的读写操作。标准 SRAM 单元具有结构简单、读写方便等优点，因此 FPGA 内部大量可编程配置信息都是依赖 SRAM 来存储的。由于它是由反相器的组合反馈组成，因此这种简单的 SRAM 单元不具备同步时序控制功能。随着半导体工艺进入纳米尺度，标准 SRAM 内部存储数据所需要的电荷量越来越少，以至于容易受到外界的干扰而改变内部数据。因此最近为了提高 SRAM 单元的稳定性需求提出了很多高可靠的晶体管级 SRAM 单元结构。对 SRAM 的详细介绍参见 2.2.2 节。图 1-15 中存储器层次结构的下一个层次是触发器。它是时序电路设计必不可少的单元。时序电路的触发器规模与逻辑状态图中的状态数目相关。触发器(D flip-flop，DFF)一般只能传输一位信号，并且需要同步时钟边沿触发，因此稳定性比 SRAM 单元要好。但是它所需要的晶体管个数要比 SRAM 单元多很多。寄存器是以字节为传输单位，可以看作是由多个 DFF 组成，却共享控制信号的多位存储单元。一般微处理器内的寄存器规模小于 4KB，速度大概在 0.25～0.5ns 之间，是微处理器指令直接操作的对象。下一个层次是高速缓冲器。通用微处理器和 DSP 芯片都有不同级别的高速缓冲器，一般通过 SRAM 的阵列方式实现。它通过分块操作进行读写，其中块大小和命中率是它的主要指标。高速缓冲器规模一般小于 16MB，速度大约为 0.5～25ns。主内存就是我们平时可以购买的内存条，一般通过 DRAM 单元实现。它的结构非常简单，一般只需要一个晶体管和电容及其刷新电路。在存储时需要刷新电容上的电荷，以防止数据因电容放电而丢失。因此它比 SRAM 单元速度慢，所需要的晶体管个数少，阵列规模要比高速缓冲器大，一般小于 16GB。DRAM 一般使用行地址和列地址复用的结构，这样可以节约管脚数目。对于常用的电脑来说，主内存一般是通过操作系统进行页面管理。触发器、寄存器、高速缓冲器和主内存都需要电源供电才能工作，一旦掉电后所有的存储数据都会丢失。磁盘的规模更大，掉电后信息依然保存，一般通过文件的方式管理。磁盘的访问速度大约为 10μm～1ms，数据传输率为 20～150MB/s。位于层次结构最底层的是磁带，一般用于备份大批量的数据，访问时间为秒级，数据传输率为 MB/s 量级，但是容量非常大。

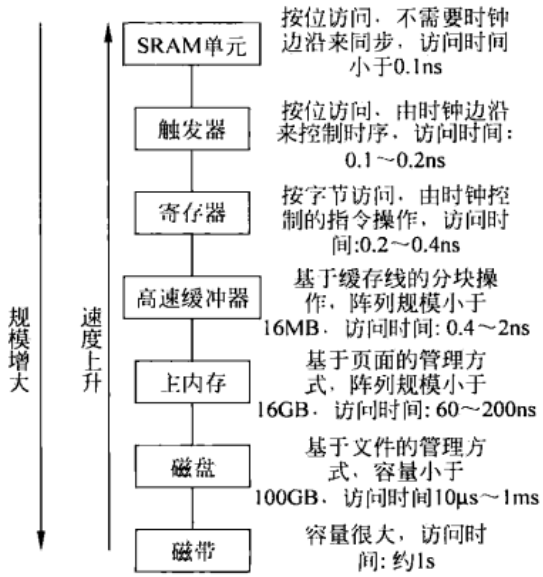


图 1-15 与通用处理器相关的存储器层次结构

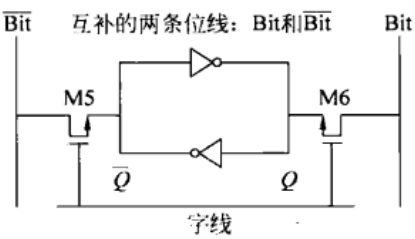


图 1-16 标准 SRAM 结构

软盘和高密度软盘(ZIP 盘)曾经是计算机的主要移动存储设备,现在基本上被基于闪存的 U 盘(USB flash disk)或者移动硬盘所代替,它们通常连接到电脑的 USB(universal serial bus,通用串行总线)接口来传送文件。除了电磁记录的存储器外,CD(compact disk,光盘)盘和 DVD(digital video disk,数字视频光盘。后来改为 digital versatile disk,为"数字多功能光盘")也很常用,它们掉电后信息仍然保存。光盘也是按照文件的方式由操作系统进行管理。和磁盘不同,光盘是通过激光在信息记录层上烧制"小坑"的方式来保存信息。对于 CD 盘,每一位小坑的表面直径大约为 1μm。读取数据时,一束激光照射到光盘表面,根据反射光的状态确定是一个坑(没有反射光)还是平面(有反射光),从而知道存储的是"0"还是"1"。DVD 有多层激光束可以聚焦的面,而且每一位所占的面积比 CD 盘格式小。另外还有一些新型的存储器,例如铁电和相变存储器,它们都具有一定的特色,但目前还没有足够的优势和磁盘、光盘竞争。从理论上来说,这些不同种类的存储单元都可以用于实现可编程性。但是由于它们处于存储层次中的不同级别,我们就可以根据它们的优越性来选择适合于 FPGA 的可编程配置存储单元。这方面的具体讨论见第 2 章的 2.2.2 节。

就如人们的大脑具有信息处理和记忆功能一样,CPU、FPGA、DSP 和 GPU 等芯片,它们的可编程性或灵活性的实现都是基于具有记忆功能的存储单元。存储单元的内容可以表示用户数据、可执行程序或 FPGA 内部的配置信息。一个二进制数本身并不具有内在的含义,它可以表示符号数、无符号数、浮点数、指令码、逻辑单元的函数功能、数据包的地址、两条导线的连接方式、不同文件的格式或其他内容等。例如对于细粒度的传统 FPGA 结构来说,一个触发器(flip-flop)只能存储一位二进制信息。硬件单元之间通过一位的时钟信号进行同步协调。对于粗粒度的通用处理器来说,一个寄存器能存储数个字节的数据,例如常用的 32 位二进制数据。算术逻辑单元和寄存器之间的数据传递是通过总线进行同步协调。这是底层硬件的计算和存储模块之间的互连与通信方式。在通用处理器提供的硬件平台上,我们可以用软件的方法实现各种高层次通信。例如国际标准化组织制定了七层 OSI (open system interconnect,开放式系统互联)模型。它包括基于物理层上面的数据链路层、网络层、传输层、会话层、表示层和应用层等。从高层的通信协议这个角度来说,我们已经不再侧重硬件信号的时序关系或者是各种总线控制器的硬件实现细节,而是各种通信协议的可靠性、传输速度和安全性等指标,尽管这些通信协议都需要底层的硬件资源作为基础。一般来说,我们把物理级底层的连接称做"互连",而高层基于协议的通信称做"互联"。虽然这两个词属同音词,但是它们之间还是有细微的差别。

下一节将讨论冯·诺依曼提出的存储程序通用微处理器信息处理技术。理解通用处理器的可编程硬件结构对掌握 FPGA 的结构和应用十分有益。

1.3　基于通用微处理器的信息处理技术

在第二次世界大战期间,由美国宾夕法尼亚大学的物理学家莫克利(John Mauchly)和工程师埃克特(Presper Eckert)组成的研究小组负责开发 ENIAC(Electronic Numerical Integrator And Computer,电子数字积分计算机),用于加速大量的数值计算。他们在 1943—1944 年期间就已经提出了"存储程序"(stored program)的概念,并计划基于这一概念开发下一代计算机 EDVAC(electronic discrete variable automatic computer,离散变量自

动电子计算机)。当时犹太人冯·诺依曼正在参与洛斯阿拉莫斯国家实验室(Los Alamos National Laboratory)的曼哈顿计划(Manhattan Project)。由于他在数学方面所具有的才能,1944 年他就参与了 ENIAC 项目的开发和后续的讨论,并且就"存储程序"的概念很快提出了技术实现方案。1945 年 6 月,冯·诺依曼完成了开发下一代计算机的技术报告,"First Draft of a Report on the EDVAC"。这份报告包含了他和莫克利、埃克特等人的讨论结果。但是当他的同事歌德斯坦(Herman Goldstine)把这份报告向欧洲和美国的同行进行广泛传阅时,报告的署名只有冯·诺依曼一人。由于这份报告的反响很大,报告中所提出的结构就被称做"冯·诺依曼体系结构",他本人后来被冠于"计算机之父"的美誉。

从计算机结构的角度来看,他们于 1946 年完成的 ENIAC 计算机虽然可以大大地提高运算速度,但是存在两大缺点:①没有存储器;②它用布线接板进行控制,甚至要搭接几天才能完成具体的任务。这样计算机的速度优势就不能得到有效发挥。而 EDVAC 计算机的主要思想是把计算程序也作为存储器内的数据自动处理,而不是通过手工操作来连接相应的电路。具体来说,在硬件结构方面,EDVAC 方案明确规定计算机由五个部分组成:运算器、逻辑控制器、存储器、输入和输出设备。如上一节所述,存储器为计算机提供了现场可编程性的硬件基础。如果存储器内部存储的不只是用户数据,还包括计算机执行的指令码,并且在计算机运行过程中这部分数据是不会改变的,那么计算机的数据和程序本身均可以进行自动编程,而不需要外面用户的干预。这也是冯·诺依曼体系结构的存储程序思想。在数据存储方面,用户数据和可执行程序都以二进制(而不是十进制)的方式存放到存储器中,这样就可以充分利用二进制逻辑电路的效率和可靠性。

从冯·诺依曼体系的硬件结构可以发现,这种通用性所付出的代价就是计算和存储在结构上的隔离,从而导致了"存储墙"等问题。换句话说,冯·诺依曼结构把运行可执行程序的 ALU 计算单元和数据存储单元在核心架构上就分开了。随着半导体工艺的进步,单个晶体管的成本急剧下降,存储单元和计算单元的硬件成本和 60 多年前相比已经变得非常便宜。但是由结构造成的存储和计算之间的通信瓶颈成为进一步提高计算效率的主要障碍。我们从第 2 章的图 2-6 就可以发现 FPGA 已经很好地突破了这种结构上的限制。即使从底层硬件电路结构优化的角度来看,由于早期限制电路工作速度的因素主要是门电路单元的延时,而不是互连延时,因此那时电路性能的优化主要侧重于单元门电路,而不是互连延时。后来随着半导体工艺技术的进步,晶体管越来越小,门延时不断下降;但是金属导线宽度越来越窄,互连延时在关键路径中所占的比例越来越高。因此电路性能的优化目标主要是如何减少电路的互连延时。后来除了延时以外,随着半导体工艺的进一步发展,电路的功耗和可靠性均成为了优化的主要目标。因此我们相信,如何改进已有的通用型 CPU 结构,进一步提高处理器的计算效率,突破计算和存储在结构上分离的限制,将对通用型处理器结构研究提出挑战。下面我们先以一个简单的例子来回顾基于通用型 CPU 的软件可编程性设计流程。读者也可参考第 3 章的 3.1.1 节来帮助理解。

图 1-17 显示了一个简单的 C 语言程序代码及其对应 Altera 公司处理器 Nios Ⅱ的汇编程序,它输出经典的"Hello world"字符串和基本的乘加运算结果。假设这个文件名为 hello_world_datapath.c,那么可以调用 C 编译器,例如 GNU C 编译器产生目标处理器的二进制可执行程序。对于 Nios Ⅱ处理器和 GCC 编译器来说,在命令行输入"nios2-elf-gcc -S hello_world_datapath.c"就会调用 Nios Ⅱ编译工具,从而把 C 代码转换为 Nios Ⅱ嵌入式

处理器的汇编指令文件。如图 1-17 所示，编译器所产生的汇编指令代码还是比较容易理解的。其中 add、addi(addition with an immediate)、stw(store a word)、mov(move)、movi(move an immediate)、ldw(load a word)分别表示“寄存器加法”、“带立即数的加法”、“保存数据”、“移动数据”、“移动立即数”、“读取数据”等操作。图中汇编代码开始部分的地址标签“.LC0”表示存放字符串“Hello world - %d\n”的首址，它是 printf 函数的第一个参数。接下去的地址标签“.main”就是 main 函数的首址，也就是 main 函数的入口。main 函数体一开始就为自己开辟了 20B 的堆栈空间，用于存放 main 函数内部的局部数据，其中“sp”、“fp”为堆栈指针(stack pointer)和帧指针(frame pointer)。然后用户代码中的局部数据“10”和“100”两个立即数被存到堆栈，接下去方框内的三条汇编指令就是用于数据计算表达式 $k=i\times 9+j$。由于这个表达式包含了一个乘法和一个加法运算，因此函数体就很自然地调用乘法(muli)和加法(add)指令来计算表达式的值。计算好结果后就把 printf 函数要输出的字符串“Hello world”和计算结果通过寄存器 r4、r5 传给 printf 库函数。这样 printf 函数就在屏幕上打印字符串和计算结果。printf 库函数返回后 main 函数就返回到调用它的系统函数。

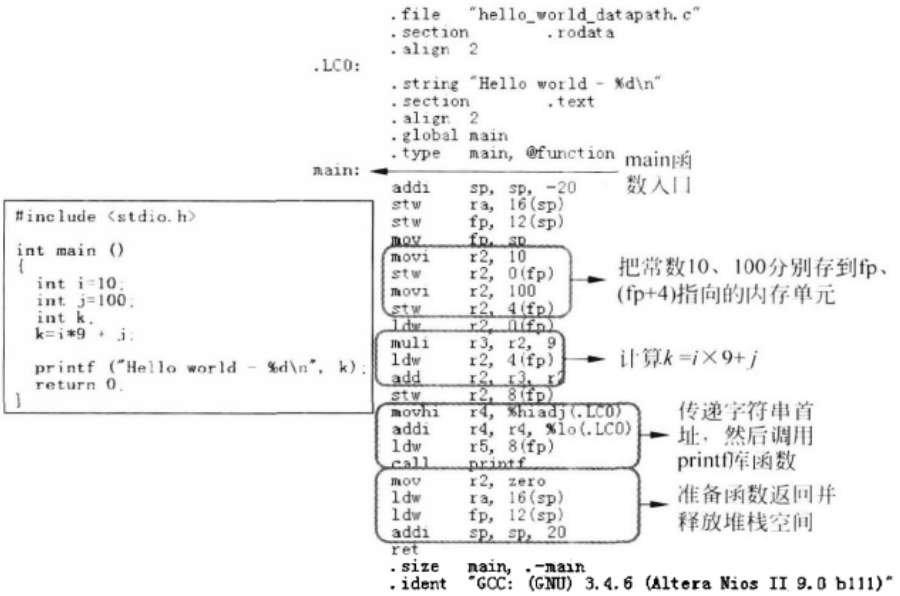


图 1-17　包含 Hello world 和简单算术运算的 C 代码及其对应的 Nios 汇编语言程序

对于编译器产生的可执行程序，可以用 objdump 等工具显示程序的指令码和汇编指令，如图 1-18 所示。图中可执行程序名字为 hello_world_datapath.elf。在一行中冒号前的数据表示指令存放的地址，冒号后的十六进制数表示指令码，最后是汇编指令助记符。我们发现在可执行程序中，编译器在 main 函数前增加了函数名为_start 的启动函数，用于初始化堆栈指针等操作，从而为 main 函数的运行做准备。在该程序中，_start 函数的入口地址为 0，main 函数和 printf 的入口地址分别为 30 和 88。我们还可以发现图 1-17 中的汇编指令和图 1-18 中的指令码严格对应，只不过连接后的可执行程序中所有的地址标签，例如“.LC0”和“.main”都已经转为内存中的物理地址了。程序运行的时候基本上就是按照指令的顺序逐条执行，除非执行一些跳转或返回指令等。

```
hello_world_datapath.elf:     file format elf32-littlenios2
architecture: nios2, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x00000000

Disassembly of section .text:

00000000 <_start>:
       0:   06c20034    movhi   sp,2048
       4:   deffc004    addi    sp,sp,-256
       8:   def6303a    nor sp,sp,sp
       c:   dec001d4    ori sp,sp,7
      10:   def6303a    nor sp,sp,sp
      14:   06800074    movhi   gp,1
      18:   d686b004    addi    gp,gp,6848
      1c:   06000074    movhi   et,1
      20:   c626c104    addi    et,et,-25852
      24:   00800074    movhi   r2,1
      28:   10a32e04    addi    r2,r2,-29512
      2c:   1000683a    jmp r2

00000030 <main>:
      30:   defffb04    addi    sp,sp,-20
      34:   dfc00415    stw ra,16(sp)
      38:   df000315    stw fp,12(sp)
      3c:   d839883a    mov fp,sp
      40:   00800284    movi    r2,10
      44:   e0800015    stw r2,0(fp)
      48:   00801904    movi    r2,100
      4c:   e0800115    stw r2,4(fp)
      50:   e0800017    ldw r2,0(fp)
      54:   10c00264    muli    r3,r2,9
      58:   e0800117    ldw r2,4(fp)
      5c:   1885883a    add r2,r3,r2
      60:   e0800215    stw r2,8(fp)
      64:   01000074    movhi   r4,1
      68:   21239a04    addi    r4,r4,-29080
      6c:   e1400217    ldw r5,8(fp)
      70:   00000880    call    88 <printf>
      74:   0005883a    mov r2,zero
      78:   dfc00417    ldw ra,16(sp)
      7c:   df000317    ldw fp,12(sp)
      80:   dec00504    addi    sp,sp,20
      84:   f800283a    ret

00000088 <printf>:
      88:   defffc04    addi    sp,sp,-16
      8c:   dfc00015    stw ra,0(sp)
```

图 1-18　用 objdump 输出可执行程序的指令码和对应的汇编指令

对于可执行程序，还可以利用编译器提供的功能阅读程序的二进制代码，也可以借用任何一个二进制文件编辑器查看其内容。图 1-19 是用 XVI32 工具打开可执行程序 hello_world_datapath.elf 文件的结果。其中左边部分是程序的二进制代码，右边是二进制代码所对应的 ASCII 字符。由于程序的指令码和 ASCII 代码没有必然关系，因此图中右边部分所显示的基本上是乱字符。但是这个文件从第二个字节开始是“ELF”的 ASCII 代码值“45 4C 46”，这表示它是一个 Linux 操作系统中常用的 ELF(executable and linkable format)格式文件。这种代码用英文称做 magic number。程序的前面部分是文件头(file header)，定义基本的文件内容信息，例如程序类型、处理器结构类型和程序头(program header)大小等。文件头是为了方便程序装入到内存执行而定义。文件头后面定义的就是指令码。例如图中第六行末尾部分，“34 00 C2 06”。它就对应于启动函数_start 的第一条指令“movhi sp，2048”的指

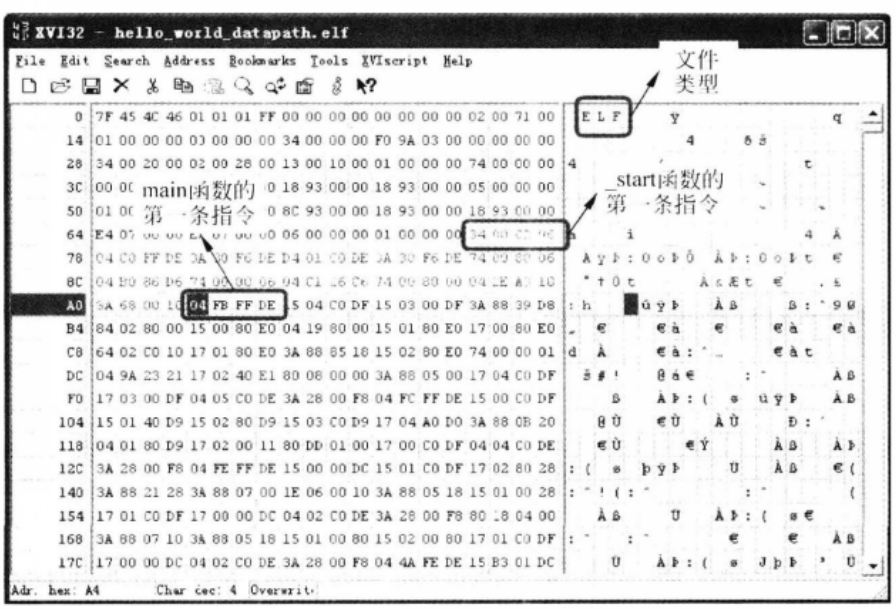


图 1-19　基于通用的二进制文件编辑器直接查看可执行文件的内容

令码“06 C2 00 34”。再接下去第九行标注位置的四个字节内容为“04 FB FF DE”，刚好对应于 main 函数的第一条指令，“DE FF FB 04”。由此可知，编辑器显示的内容和汇编指令代码一一对应，不允许有任何的差错。需要注意的是，由于 Nios Ⅱ程序代码采用的是 little endianess(小尾数法)，因此字节次序刚好和编辑器的显示次序颠倒过来，即指令码“06 C2 00 34”从低字节往高字节显示的次序就是“34 00 C2 06”；同样，main 函数的第一条指令“DE FF FB 04”在通用的编辑器中显示为“04 FB FF DE”。读者可以逐条比较所有的指令，会发现它们和编辑器显示的内容完全一致，没有任何差错。

由以上讨论可见，通用型 CPU 的可编程性实现主要是基于以下思想：首先信息处理过程，即计算过程被划分为简单的算术和逻辑运算，包括加法、乘法、与、或等基本运算。这些基本运算就用硬件上的 ALU 单元来实现。其次计算所需要的数据和计算结果是通过总线传输到存储器和寄存器，这些数据通过分时共享 ALU 计算单元进行处理。因此在运行时 CPU 就需要取指、译码、执行和结果保存等基本的流水线过程。最后，程序和用户数据都是共享一条总线。随着计算单元速度不断加快，总线的数据传输速度就成为了束缚 CPU 计算效率的瓶颈。因此在 CPU 的结构上引入了多种解决方法来突破这个瓶颈。例如 DMA (direct memory access，直接存储访问)技术能避免占用 CPU 资源，实现外设和内存之间的直接数据传输。DMA 方法能够减少 CPU 的压力，提高了总线传输大量数据的效率。另一种方法就是 1.4 节要讨论的 DSP 技术，在硬件结构上引入了两条总线，让程序和数据分别占用不同的传输通道，从而缓解了数据传输瓶颈问题。还有一种方法就是在通用计算机结构上引入单指令多数据 SIMD 的并行工作方式。这样一条指令可以对多个不同的数据同时进行操作。这些结构上的改进都能提高通用型处理器的计算效率。但是在 1.6 节和第 2 章的讨论中，我们可以发现 FPGA 在结构上提供了一种新型的并行方法，从另外的途径提高了信息的处理效率。

1.4 DSP 技术及其应用

类似于通用微处理器，DSP 处理器也是基于存储程序结构进行改进。针对 DSP 应用领域，例如语音或图像、雷达信号、生物医学、仪器仪表等对信号处理实时性和稳定性要求较高的应用领域，通用微处理器的实时性能和功耗指标往往达不到要求，因此它不适合于复杂的数字信号处理。另外，虽然 FPGA 具有很强的数据并行计算能力，但是相比于专用的 DSP 芯片，FPGA 的缺点是成本高、功耗大、使用不方便。因此相对通用微处理器和 FPGA 来说，DSP 处理器自从 20 世纪 80 年代初诞生以来发展迅速，其主要特点是采用程序和数据存储器分开访问的哈佛结构，让处理器可以同时访问指令和数据存储器。另外，DSP 处理器还具有专门的硬件乘法器和一些特殊的 DSP 指令，便于快速实现各种数字信号处理算法。这些特点使得它能在一个指令周期内就能完成乘法和加法运算。简单的 DSP 应用也可以在单片机上实现，例如 MCS-51 等。不过对于复杂的 DSP 应用，FPGA 通常作为协处理器使用，以进一步提高系统的灵活性和性价比。有些 DSP 芯片具有嵌入式通用微处理器，例如 TI 公司的 OMAP(open multimedia applications platform，开放式多媒体应用平台)便将 DSP 核和 ARM 核集成在一起构成一个双核芯片，能够满足 3G 无线应用的需求。

图 1-20 为一个典型的 FIR(finite impulse response，有限冲击响应)滤波器的结构框图，

其中 N 为滤波器的阶数。各延时单元，Z^{-1}，在硬件上就由寄存器来实现。滤波器的输出表达式为

$$y(n) = \sum_{i=0}^{N-1} a_i x(n-i) \tag{1-16}$$

根据以上公式，FIR 滤波器的输出值需要很多的乘法器和累加器参与计算。它们先把输入数据 $x(n-i)$ 和滤波器的系数 a_i 相乘，再把各个积项在累加器中相加，然后实时输出累加器的结果 $y(n)$。对于这样的应用需求，通用微处理器一般不具备并行工作的乘法器和累加器，因此需要 DSP 处理器来提高计算效率。下面定量分析 DSP 处理器和通用微处理器实现 FIR 滤波器的时钟周期数差异。

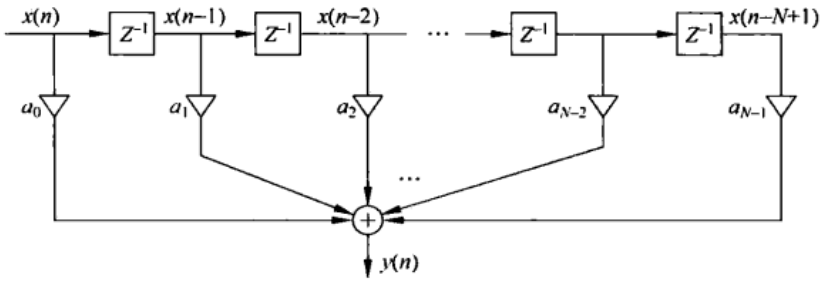


图 1-20　典型的 FIR 滤波器框图

由图 1-20 可见，FIR 滤波器的基本内存操作在单核通用处理器上起码需要执行以下 4 个步骤：

(1) 读入指令码，表示要执行乘加操作；

(2) 读入新的采样数据 $x(n)$；

(3) 读入滤波器的系数 a_i，$0 \leqslant i \leqslant N-1$，并执行乘加运算；

(4) 把 $x(n)$ 写到下一个延时采样数据 $x(n-1)$。

根据图 1-21(a)中通用微处理器的冯·诺依曼结构，由于程序和数据存放于同一块存储器中，每一个时钟只能读取或写入一个指令码或数据，所以以上 4 个步骤需要 4 个时钟周期来完成。对于 DSP 处理器中常用的哈佛结构，可以通过程序和数据总线同时读取指令码和数据，因此 FIR 的基本内存操作可以在第一个时钟读取指令码和 $x(n)$；然后在第二个时钟读取系数 a_i。但是写 $x(n-1)$ 还需要一个时钟，因此对于哈佛结构，以上 4 个步骤需要 3 个时钟周期来完成，见图 1-21(b)。

为了进一步提高计算效率，还可以对存储器结构进行进一步的改进，采用如图 1-21(c)所示的改进型哈佛结构。在该结构中，存储器 A 和存储器 B 均可以存放程序或数据。对于这样的结构，FIR 滤波器的基本内存操作可以在第一个时钟读取指令码和 $x(n)$；然后在第二个时钟周期读取系数 a_i 并且把 $x(n)$ 写到下一个延时采样数据 $x(n-1)$。这样就可以在两个时钟内完成以上步骤，前提是只要把程序和系数 a_i 放在同一块存储器。

针对 DSP 领域中常用运算的高性能需求，例如数字滤波、卷积和傅里叶变换等，就有了一些专用的 DSP 处理器。相比通用型 DSP 芯片，它们是专门为特定的 DSP 运算设计的，因此对于这些特定运算的处理效率就很高。例如 Motorola 公司的 DSP56200 就是适合于自适应滤波的定点专用型 DSP 处理器。

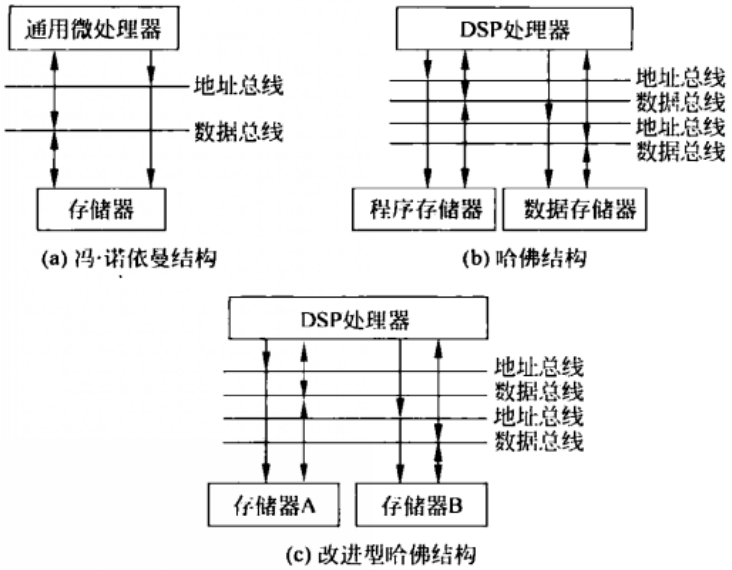
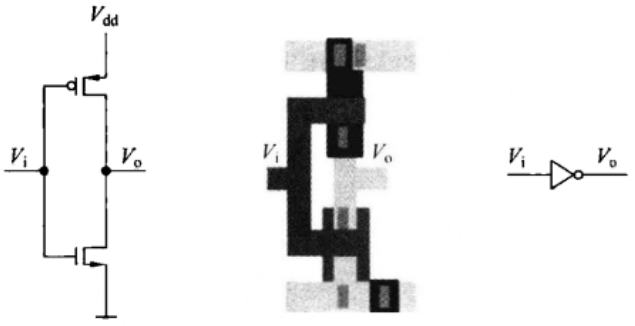


图 1-21　处理器和存储器的不同结构

1.5　专用数字集成电路设计

DSP 处理器的成功，主要归功于它能有效地解决通用处理器在数字信号处理应用领域效率低下的问题。但是除了数字信号处理领域之外，还存在很多的专用领域。对于这些专用领域来说，如果采用通用处理器进行计算同样会面临效率低下的问题。于是从 20 世纪 80 年代开始，专用数字集成电路应运而生。

从硬件电路的角度来说，不管一块数字芯片多么复杂，底层都是通过晶体管来实现逻辑功能的。图 1-22 所示为最基本的 CMOS 反相器电路、对应的版图和逻辑符号。它由一个 PMOS 管和一个 NMOS 管组成。当输入为 1 或“高电平”时（除非另外说明，本书采用正逻辑表达方式，即“1”表示逻辑高电平；“0”表示逻辑低电平），下面的 NMOS 管导通，输出 V_o 就相当于接地。上面的 PMOS 管截止，隔断了输出端 V_o 与电源正极的连接。反之，当输入为 0 时，下面的 NMOS 管截止，输出 V_o 就和地线断开。PMOS 管导通，输出 V_o 就连到电源正极。因此，输出信号 V_o 是输入信号 V_i 的反相。由于反相器在输出“0”和“1”时都能直接



(a) 晶体管级电路　(b) 版图　(c) 逻辑门符号

图 1-22　CMOS 反相器

到电源输出端或地，因此它能对输入信号进行很好的方波整形。

在软件设计领域，汇编语言早已经不适合于编写大程序。虽然用汇编语言编写的程序的执行效率很高，但是大程序一般都用高级语言，例如 Fortran、C/C++、Java、C＃等语言来编写。20 世纪 80 年代，基于晶体管和逻辑门的设计方法不能满足大规模集成电路的需求，于是 Verilog 或 VHDL 作为硬件描述语言便应运而生。基于 HDL(hardware description language，硬件描述语言)的数字电路设计就好像编写软件代码一样灵活方便。当然写好的代码需要经过逻辑综合工具产生门级网表，然后经过后端布局布线工具完成后端版图的设计。

图 1-23 显示了两位加法器的 Verilog 代码，可以发现它和 C 语言的语法非常接近。第一行 module 所定义的是模块的名字“adder2”，类似于 C 语言中的函数名。接下去是模块的输入和输出管脚名。“sum＝a＋b”表示把输入 a 和 b 的结果相加，然后送给输出端 sum。如果需要设计 8 位或其他宽度的加法器，只需更改输入和输出管脚宽度的数字即可。图 1-23 的右半部分为 Synopsys 公司逻辑综合工具 Design Compiler 输出的两位加法器的门级电路。它把左边的 Verilog 代码综合成用基本逻辑门实现的电路，其中包括三个异或门和一个与门。

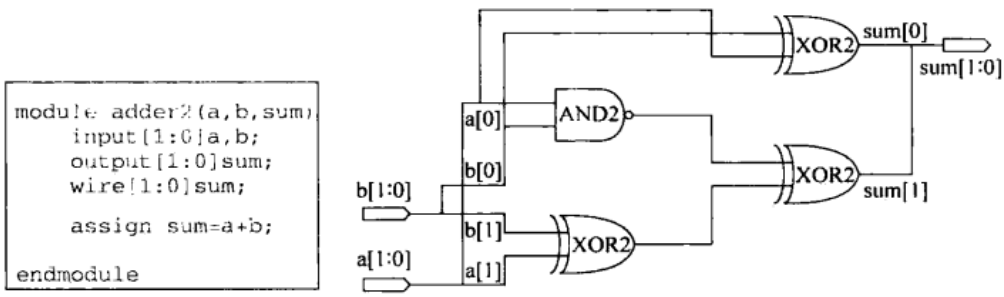


图 1-23 两位加法器的 Verilog 代码和综合后输出的门级电路

从 20 世纪 80 年代开始，逻辑综合和布局布线工具为专用集成电路的设计与普及提供了必要的条件。当时逻辑综合和布局布线工具的首要目标是提高电路的时序性能，这就是图 1-24 所示的“版本 1.0”所面临的挑战，就是要实现时序驱动的综合和布局布线流程(timing driven flow)，并且能够使逻辑综合和布局布线流程有效地连接在一起从而构成一个整体流程。随着半导体工艺进入深亚微米水平，特别是进入 0.25μm 工艺之后，前端逻辑综合工具和后端的布局布线工具在时序性能上往往无法一致。换句话说，前端综合时确定的关键路径和后端布局布线后的关键路径往往不同。从时序性能优化的程度来说，前端综合工具对时序性能的影响往往要大于后端布局布线工具的影响。为了使后端布局布线后的时序性能达到用户的要求，就需要用户在前端工具和后端工具之间多次反复。这就是图 1-24所示的“版本 2.0”所面临的挑战，其主要问题就是要减少前后端工具之间不断调整电路结构和修改设计约束的次数。具体地说，当前端综合优化的电路在后端布局布线时发现时序约束不能满足，需要重新调整前端工具的时序约束和优化目标，再重新进行布局布线。如果新的布局布线结果仍旧不满足用户要求，那么又要在综合工具中进行修改，直到重新布局布线后的电路满足时序性能要求。随着半导体工艺的进一步更新换代，芯片设计流程还需要不断改进。例如图 1-24 中解决电路规模与复杂度问题的层次化设计流程

(hierarchical flow,HF),解决信号完整性(signal integrity,SI)问题的设计流程,解决电路功耗问题的低功耗设计和功耗管理流程。当半导体工艺达到65nm的时候,设计流程还需要解决可制造性(design for manufacturability,DFM)问题,保证设计出来的芯片能够被正确制造,且有较高的成品率。

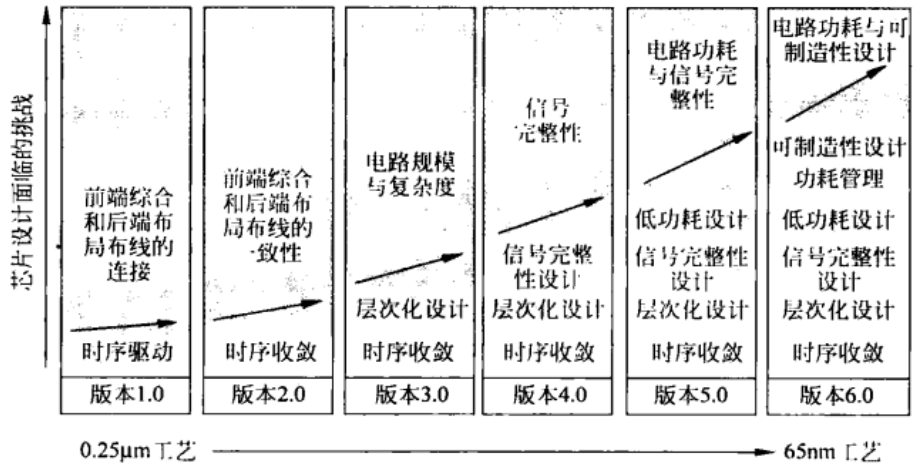


图 1-24 TSMC公司在2005年设计自动化国际会议(DAC)上介绍的芯片设计流程的演变

1.6 系统级FPGA计算平台的特点

在早期的可编程逻辑器件中,现场可编程性主要是以存储器的形式来实现的,例如可编程只读存储器(PROM)、紫外线可擦除只读存储器(EPROM)和电可擦除只读存储器(EEPROM)均可用于可编程性的实现。由于结构的限制,它们只能完成简单的数字逻辑功能。为了提高可编程芯片的逻辑功能,20世纪70年代中期出现了一类结构上直接由"与"门和"或"门阵列组成的可编程逻辑器件(PLD),它能够以乘积项之和(sum-of-product,SoP)的形式完成大量的组合逻辑功能。这一阶段的产品主要有可编程阵列逻辑(programmable array logic,PAL)器件和通用阵列逻辑(generic array logic,GAL)器件。现场可编程性的实现工艺有反熔丝(anti-fuse)技术、EPROM技术和EEPROM技术。由于GAL具有可编程的逻辑宏单元结构,它能够取代大部分SSI、MSI和PAL器件。20世纪80年代GAL器件在民用电子和军工电子中都得到广泛应用。

80年代中期,美国的一些集成电路公司分别推出了类似于PAL结构的CPLD器件(complex programmable logic device,复杂可编程逻辑器件),即把多个简单的PLD结构和层次化互连资源集成到一个芯片中去,使其具有更大的数字信号控制和电平转换能力,非常适合作为胶合逻辑(glue logic)应用。但是它的数据计算能力明显落后于当时普遍采用的数字微处理系统。可编程逻辑器件传统上之所以被作为"胶合逻辑"使用,主要是由于在一块芯片上还不能集成一个数字处理系统所有功能的前提下,系统中的每个功能模块只能由各自的芯片来完成。但是这些芯片的接口往往不能直接相连。可编程器件的IO接口十分灵活,可以根据用户的需求配置成不同的格式。因此,构成一个数字处理系统的多块芯片可以通过一块可编程逻辑器件"粘合"在一起,从而很方便地构造了一个数字系统。20世纪

80年代典型的数字微处理器系统包含微处理器、存储器和一些特殊功能的中小规模逻辑器件。为了弥补可编程芯片在数据计算能力上的缺陷，1985年美国Xilinx公司推出了世界上第一款FPGA产品，它包括两个可编程器件XC2064和XC2018以及支持布局布线的软件设计工具。从硬件结构上看，FPGA器件采用了基于LUT的结构。它比早期可编程逻辑器件的与/或门阵列结构具有更强的数据处理能力。另外FPGA芯片内部包含了更多的触发器和复杂的互连资源，可以方便地实现各种复杂的流水线型数字信号处理系统。与专用芯片相比，FPGA器件具有设计开发周期短、NRE(non-recurring engineering，一次性工程费用)成本低、设计灵活、友好的用户集成开发环境等优点，因而被广泛应用于电子产品的原型验证和中小量产规模的电子产品设计。

经过20多年的发展，可编程逻辑器件所采用的半导体工艺水平从早期的2μm发展到现在的65nm技术；规模从不到1000门的XC2064，只有64个可编程逻辑单元、8万多个晶体管发展到千万门级Virtex5，具有33万个可编程逻辑单元、超过10亿个晶体管；应用领域从早期的胶合逻辑，扩展到算法逻辑设计、专用芯片的原型设计，和数字信号处理加速以及嵌入式系统领域等。因此，我们把器件内部直接包含了众多计算模块、存储模块和可编程逻辑单元，从而能够快速搭建一个数字处理系统的可编程逻辑阵列芯片称为系统级FPGA。系统级FPGA在逻辑密度、速度、功耗、成本等方面都有巨大的突破，它有以下特点。

(1) 随着半导体集成电路的技术发展和不断上升的计算性能应用需求，多核结构的单芯片结构已经成为必要的解决途径。从硬件结构上来说，现代商用FPGA器件是集成了多种计算粒度IP核的异构多核芯片。系统级FPGA一般具有片内微处理器、多种模式的存储器、宽度和精度灵活可调的乘加器DSP模块、支持各种标准的IO模块、PLL(phase locked loop)或DLL(delay locked loop)等时钟产生与管理模块，能够满足系统级应用的综合需求。系统级FPGA的典型硬件结构如图1-25所示。对于存储器资源需求较大的应用来说，还可以利用细粒度的LUT单元阵列构建分布式存储器，以扩大片内存储单元的容量。这种采用细粒度LUT构建粗粒度存储器的结构一般称为分布式存储结构，这主要是由于这些LUT是由分布在不同的可编程逻辑单元组成。但是针对DSP等应用领域，现在的系统级FPGA器件还没有嵌入式的A/D或D/A转换模块。

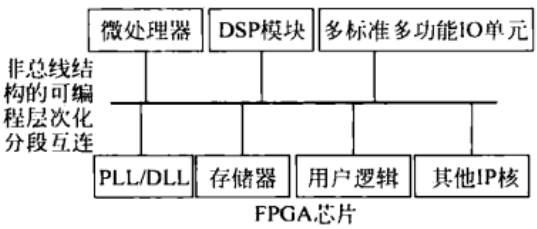


图1-25 系统级FPGA典型硬件结构

(2) 为了降低深亚微米芯片的功耗，最近推出的系统级FPGA器件在硬件结构和软件方面都提供了低功耗设计的特性。在硬件单元结构上提供了低电压工作的低功耗模式，在软件算法方面支持低功耗设计的优化算法。为了克服传统上高功耗的缺点，FPGA已经迈出了一大步，使之能够适用于消费类的终端用户产品。

(3) 系统级FPGA软件除了支持经典的RTL级硬件描述文件的编译外，还提供完善的可重用IP库，和针对嵌入式微处理器应用的C/C++高级语言的编译和调试环境。面向DSP应用，系统级FPGA还提供和Matlab工具配套的数字信号处理系统的开发与调试。但是相对于成熟的软件调试和DSP系统开发工具，系统级FPGA的系统开发工具链还需要

继续完善。

（4）支持动态可重构技术。FPGA 是现场可编程器件，一般只作为静态配置使用，即在 FPGA 运行的时候，其配置信息一直是保持不变的。由于系统级 FPGA 具有嵌入式微处理器并可以运行嵌入式操作系统，因此在 FPGA 运行的时候，可以根据系统需求通过操作系统的动态调度对 FPGA 进行局部配置下载。对一个系统应用来说，不是所有的子模块都是保持运行状态的。因此 FPGA 在某一个时刻把部分硬件资源配置成为功能子模块 A，在另一个时刻配置成功能子模块 B，这样可以达到充分利用硬件资源的目的。虽然目前还没有成熟的实用工具，但是一些产品如 Xilinx 公司的 FPGA 器件能够支持运行时局部下载，为实现动态可重构技术提供了很好的基础。

1.7　本书结构

本书内容跨度大，从硬件可编程技术、可编程硬件资源结构到基于 FPGA 的数字电路设计和软硬件协同的系统级应用设计，它们之间的关系如图 1-26 所示。在第 1 章的引言后，本书第 2 章开始介绍各种常用的硬件可编程技术，例如 EEPROM、反熔丝和熔丝、SRAM、Flash 技术等。可编程性就是通过这些硬件可编程存储单元来实现的。这些可编程单元在硬件上控制可编程逻辑单元、互连资源、输入输出和各种粗粒度的系统级 FPGA 硬件资源等，因此第 2 章主要介绍硬件资源结构。可编程技术是硬件结构和软件编程之间的桥梁。控制各种可编程硬件资源的可编程数据是通过下载细粒度的位流文件和粗粒度的可执行文件来完成，其中位流文件主要控制细粒度的可编程逻辑单元、互连资源、输入输出单元和粗粒度单元的配置模式，如 DSP 模块和存储器的数据宽度等。粗粒度的可执行文件主要是下载到嵌入式的微处理器中运行，因此第 3 章主要介绍基于 FPGA 的数字电路设计流程。用户编写的各种硬件语言描述代码，例如 VHDL（very-high-speed integrated circuit hardware description language，高速集成电路硬件描述语言）或者 Verilog 代码，以及电路原理图描述文件等，经过 FPGA 的编译流程就产生了所需要的位流文件。这个位流文件通过编程下载工具下载到 FPGA 芯片内部就实现了用户的电路功能。第 4 章和第 5 章分别

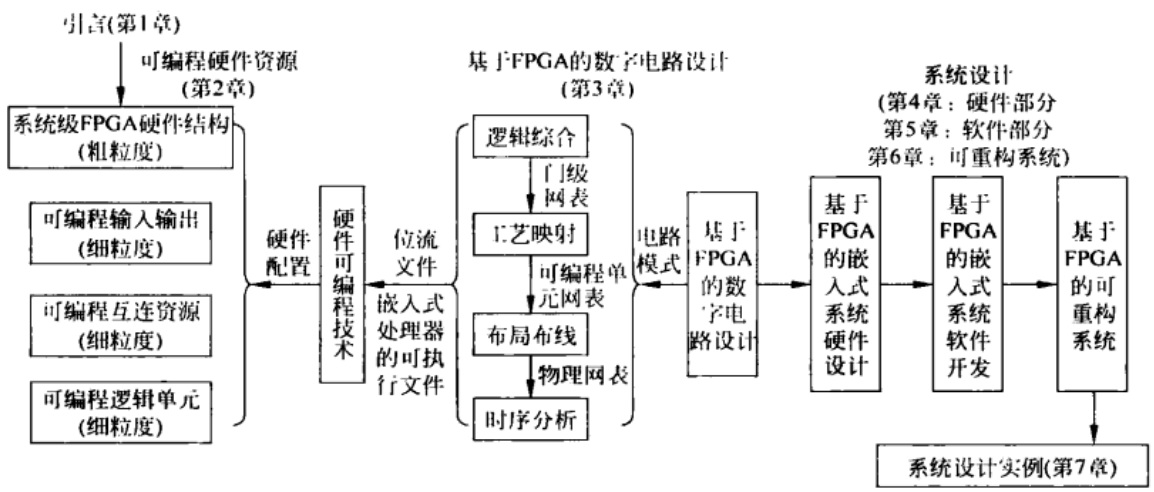


图 1-26　本书内容结构关系图

介绍嵌入式系统的硬件和软件设计方法，主要介绍各种常见的微处理器、片上总线和自定义外设电路的设计方法和嵌入式系统软件开发技术。第 6 章主要介绍基于 FPGA 的可重构系统及其设计方法，从可重构计算的定义、可重构系统及其分类出发，以部分动态重构系统为研究对象介绍模块化的可重构系统设计方法及其设计流程。第 7 章通过一个嵌入式系统设计实例对前面各章所学到的知识进行应用。本书附录部分还提供了一些上机材料。

习题

1. 假设一幅电视画面的像素为 800×600，每个像素的灰度等级为 10，试计算一幅电视画面的平均自信息量。

2. 利用式(1-9)证明对于均衡概率的 n 个信息，它们的平均信息量等于 $\log_2 n$，从而至少需要平均 $\lceil \log_2 n \rceil$ 位二进制数才能表示这些信息，其中 "$\lceil x \rceil$" 表示刚好比 x 大的整数值。

3. 根据 1.1.4 节图 1-12 所示的简化产品销售模型，计算当 $T=30$，$D=12$ 时延时上市产品的理论损失比率。

4. 画出标准 SRAM 单元和 DFF 触发器的晶体管级电路，并在面积和速度、稳定性方面做比较。

5. 画出电磁存储器的层次化结构，并说明这些存储器的特点和应用场合。

6. 描述通用处理器和 DSP 处理器在硬件结构、数据格式和软件调试工具上的差别。

7. 列举系统级 FPGA 的基本特点，并以商用产品举例说明。

参考文献

[1] Nicholas Negroponte. Being Digital. Alfred A. Knopf, Inc. 1995；胡泳，范海燕. 数字化生存. 海南：海南出版社，1996

[2] EETimes. Survey Americans find cellphones hardest to give up. www.eetasia.com/ARCH_2008_3_11_20.HTM

[3] Thomas L Friedman. The World Is Flat: A Brief History of the Twenty-first Century. 3rd edition, Picador, 2007；何帆，肖莹莹，郝正非译. 世界是平的——21 世纪简史，第 2 版. 长沙：湖南科学技术出版社，2006

[4] Francis A Schaeffer. How Should We Then Live? The Rise and Decline of Western Thought and Culture. Fleming H. Revell Company；梁祖永等译. 前车可鉴：西方思想文化的兴衰. 北京：华夏出版社，2008

[5] 王阳元，王永文. 我国集成电路产业发展之路——从消费大国走向产业大国. 北京：科学出版社，2008

[6] 工业和信息化部网站，http://www.miit.gov.cn/

[7] Kurt Godel. On Formally Undecidable Propositions of Principia Mathematica and Related Systems. New York: Dover Publications, Inc., 1992

[8] Raphael Sanzio，雅典学院，http://www.asds.org/2005A/G/pg1.htm

[9] Boole George. An Investigation of The Laws of Thought on which are founded the mathematical theories of logic and probabilities. New York: Dover Publications, Inc., 1958

[10] Goser K, Glosekotter P, Dienstuhl J 著. 陈贵灿等译. 纳电子学与纳米系统——从晶体管到分子与量子器件. 西安：西安交通大学出版社，2006

[11] Shannon C E. A Methematical Theory of Communication. Bell System Technical Journal, 1948, 27: 379-

423, 623-656

[12] Ernest Nagel, James R Newman. Godel's Proof, Revised Edition. New York: New York University Press, 2001; 陈东威, 连永君译. 哥德尔证明. 北京: 中国人民大学出版社, 2008

[13] 吾民. 读大学不只为找工作. 南方周末, E30 版, 2009 年 6 月 25 日

[14] Douglas R. Hofstadter, Godel, Escher, Bach: an Eternal Golden Braid, Penguin Books, 1980, reprinted with a preface to the 20th anniversary edition 2000; 乐秀成译. GEB ——一条永恒的金带. 成都: 四川人民出版社, 1984; 哥德尔·艾舍尔·巴赫——集异璧之大成. 北京: 商务印书馆, 1996

[15] Will Durant. The Story of Civilization Ⅲ- Caesar and Christ. New York: Simon and Schuste, 1980

[16] Shannon C E. A Symbolic Analysis of Relay and Switching Circuits. Trans. A. I. E. E., 1938, 57: 713-723

[17] John Horgan. Claude E. Shannon. IEEE Spectrum, 1992, 29(4): 72-75

[18] Alexander Wolfe. Intel Clears Up Post-Tejas Confusion. VARBusiness, 2004-5-17 (http://www.varbusiness.com/sections/news/breakingnews.jhtml? articleId=18842588)

[19] BBC News. Quantum computer slips onto chips, 2009-9-4

[20] O'Brien J L, Pryde G J, White A G, et al. Demonstration of an all-optical quantum controlled-NOT gate. Nature, 2003, 426(6964): 264-267

[21] 赵千川译. 量子计算和量子信息. 北京: 清华大学出版社, 2004; 英文版: Nielsen M A, Chuang I. Quantum Computation and Quantum Information. Cambridge: Cambridge University Press, 2000

[22] 李宗荣. 理论信息学: 概念、原理与方法. 华中科技大学系统科学研究所博士论文, 2004

[23] 田丽华. 信息论、编码与密码学. 西安: 西安电子科技大学出版社, 2008

[24] Mark D Birnbaum. Essential Electronic Design Automation (EDA). Pearson Education, Inc., 2004

[25] Gordon E Moore. No Exponential Is Forever: But "Forever" Can Be Delayed, IEEE International Solid-State Circuits Conference (ISSCC), San Francisco, 2003, PDF 文件可以从 Intel 的网站下载 (www.intel.com)

[26] 上海市信息化委员会. 2007 年上海集成电路产业发展研究报告. 上海: 上海教育出版社, 2007

[27] 工业和信息化部网站, http://www.mii.gov.cn/art/2008/01/09/art_3069_35787.html

[28] Frank Vahid, Tony Givargis. Embedded System Design: A Unified ardware/Software Introduction San Francisco, John Wiley & Sons, 2002

[29] ITRS 报告, http://www.itrs.net/Links/2001ITRS/ExecSum.pdf, 2001: 4-7

[30] Danny Biran. Reduce Cost, Risk and Time-to-Market with an FPGA-to-Structured ASIC Strategy, http://www.cotsjournalonline.com/home/article.php? id=100503, May 2006

[31] 霍华德·里奇著. 郑志丰译. 《时间简史》导读. 长沙: 湖南科学技术出版社, 2006

第 2 章

系统级 FPGA 硬件结构

2.1 PLD 和 FPGA 的整体结构

PLD(可编程逻辑器件)和 FPGA(现场可编程门阵列)均属于现场可编程逻辑器件,即用户在使用这些器件的现场就可以通过可编程可配置的方式实现不同的电路功能。PLD 和 FPGA 是相对于专用芯片来说的。专用芯片一旦制造完成后它的功能就不能改变。PLD 器件传统上是指基于可编程与或阵列的结构;而 FPGA 是指基于 LUT 的阵列结构。但是由于商业利益或专利的原因,本书在讨论过程中所指的可编程逻辑器件避免区分 PLD 和 FPGA,除非另外说明。

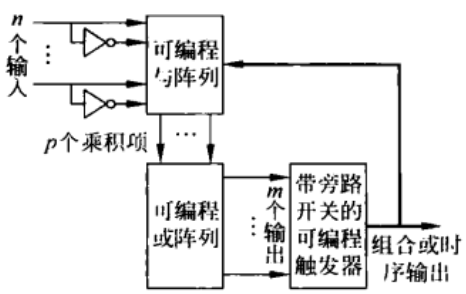


图 2-1 传统 PLD 可编程单元的基本结构

从技术角度来说,传统意义上的 PLD 逻辑单元一般是由可编程与、可编程或阵列组成,其基本结构如图 2-1 所示。由于与、或运算配合取反的输入就可以构成完备运算集,PLD 能够实现任意的组合逻辑函数。因此只要与或阵列的规模足够大,理论上就可以实现任意复杂的组合函数。一个 PLD 器件就是可编程与或逻辑单元和多个可编程触发器的阵列化结构实现。图 2-1 中的可编程触发器可以配置成不同的置位、复位、时钟触发边沿等多种工作模式。可编程与或阵列和可编程触发器都包含多个输出,即可以同时实现多个逻辑函数功能。这种单元内部有可编程旁路开关,通过跳过触发器直接输出组合逻辑函数,就能利用多个逻辑单元实现更大规模的组合电路。可编程触发器的部分输出还可以反馈到单元内部的可编程与阵列,用于快速实现简单的有限状态机。如果把可编程与或阵列的输入和输出与触发器的输出和输入端相连,就能够实现各种时序电路。PLD 内部的可编程数据传统上是存储在非挥发性器件中,因此掉电后可编程数据依然保存在器件内部,上电后无须从外面读取可编程信息。

虽然理论上来说 PLD 能够实现任意的逻辑函数,但是器件内部的可编程阵列需要足够多的二维可编程开关,才能保证函数实现的通用性和灵活性。另外,由于组合电路优化算法依赖于可编程阵列内部开关的拓扑结构,并且多个函数输出之间还需要考虑乘积项的共享以节约硬件资源,因此相应的电路优化算法比较复杂[1]。有时由于硬件电路结构的限制而无法达到更有效的优化目标。为了克服传统 PLD 器件的这些缺点,美国 Xilinx 公司于

1985年推出包含64个可编程逻辑单元的FPGA现场可编程器件。和传统的PLD结构相比，它所能实现的电路功能强大，组合电路的优化算法简单。FPGA一般采用基于LUT的可编程逻辑单元结构。一个n位输入的查找表只要把函数的真值表下载配置到LUT内部，就能够实现2^{2^n}个布尔函数，因此基于LUT的函数实现非常灵活方便。出于工艺先进性与兼容性考虑，主流FPGA的可编程信息一般是通过SRAM单元来存储，因此掉电后可编程数据也就消失，上电后需要先从片外读取可编程数据后进行编程下载，才能实现电路的正常功能。Altera公司在成功推出MAX系列的PLD产品约15年后，推出了称之为MAX Ⅱ系列的可编程器件。从技术上来说，MAX Ⅱ器件的核心部分是基于LUT和SRAM结构的FPGA器件结构，它内部嵌入了非挥发的Flash存储器。所以从用户的角度来说，用户的可编程信息甚至部分的用户数据可以加载到片内的Flash存储器，达到PLD器件所具有的非挥发可编程数据的目的。从这个意义上来说，Altera公司称为MAX Ⅱ的CPLD器件已经不再区分传统上PLD和FPGA器件核心结构的差别。

比较传统的PLD和基于LUT的FPGA结构，我们可以发现前者的可编程性主要是通过控制与或阵列的连接关系来实现，即“与”、“或”逻辑门本身是不可编程的。而后者的LUT实际上就是细粒度的存储单元，即对于给定的地址（输入变量值），都可以返回一位数据（函数输出值）。因此LUT逻辑功能的可编程性要比“与”、“或”阵列的连接可编程性更为强大。LUT内部的晶体管不再像“与”、“或”阵列内部的晶体管一样实现固定的与或逻辑。从这个意义上来说，如果把LUT看成是细粒度的存储单元，那么LUT就是利用数据存储属性来实现逻辑计算，这种思想和冯·诺依曼等人提出的“存储程序计算机”非常类似。只不过细粒度的单元更容易阵列化，即一个芯片内可以包含成千上万个逻辑单元，从而具有很高的硬件计算并行性。

FPGA的基本结构如图2-2所示，图中逻辑单元阵列规模为3×3，即包含三行三列可编程逻辑单元。这些逻辑单元内部由多个可编程查找表和触发器组成。可编程逻辑单元之间为互连资源，图中只画出了包含3条互连线段的开关盒（switch block）和连接盒（connection block）结构。实际FPGA器件的互连结构要复杂得多，一般包含多种线段长度、条数、方向和驱动能力的可编程连线。连接盒主要用于把逻辑单元的输入输出端口连接到互连资源，而开关盒主要是在水平和竖直的互连线之间进行连接切换。可编程逻辑单元和可编程互连资源构成了FPGA的核心部分。FPGA的四周为可编程输入输出（input/output）单元。这些单元内部包含了多个基本的输入输出端口，它们可以被配置为输入、输出、双向、带寄存器和不带寄存器等多种工作模式，并且支持不同的电平标准和接口协议。因此FPGA的输入输出单元结构一般要比专用芯片的输入输出单元复杂很多。为了提高输入输出单元和内部逻辑单元与互连资源的连通性，芯片四周专门有针对于IO单元的连接盒、开关盒和连线。其中连接盒用于连接IO单元和互连资源，而开关盒用于IO互连资源内水平和竖直互连线段之间的切换。当互连线长度超过多个可编程逻辑单元长度时，开关盒内部也包含不同数量的直通线段。从芯片的面积来看，FPGA内部的互连资源占据了大部分的版图面积，这也可以直观地从图2-2中看出，开关盒和连接盒的数量和所连接的线段数目都要远远超过可编程单元的数量和端口数目。

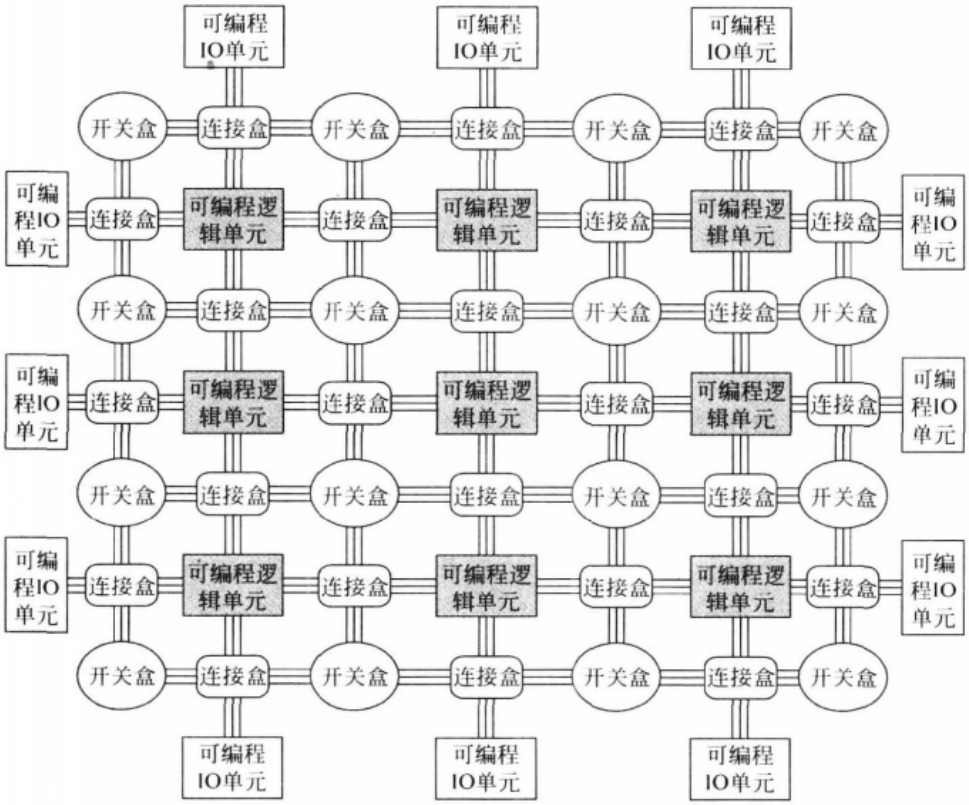


图 2-2 传统 FPGA 器件的基本结构

2.1.1 传统 PLD 器件的单元结构

基于图 1-3 对信息时代历史发展的理解，19 世纪布尔把严格的数学推理方法引入到逻辑思维领域，从而诞生了一门新的数学分支——布尔代数。它采用二值逻辑，而不是我们已经习惯的初等代数中的十值逻辑。从硬件实现的角度来看，二值逻辑只要求控制硬件中的两个状态，例如开关的导通和截止、电压的高电平和低电平等，因此二值逻辑对应的硬件电路十分简单、可靠，且具有高速度、低成本等优点。而十值逻辑需要硬件上提供十个离散的状态，并且在硬件上能够在各个状态之间方便切换，因此硬件电路实现非常复杂，并且运行不可靠，非常容易受噪声影响。如 1.3 节所述，EDVAC 计算机设计时就抛弃了它的前身 ENIAC 计算机所采用的十进制数据表达方式，而是采用了硬件更容易实现的二进制逻辑。目前主流的数字电路均采用硬件容易实现的二值逻辑。

根据布尔代数的基本原理，任何一个逻辑函数都可以基于“与”、“或”、“非”三种基本运算进行规范展开并优化，如下式所示：

$$\begin{aligned} f(A,B,C) = {} & a_0 \cdot \overline{A} \cdot \overline{B} \cdot \overline{C} + a_1 \cdot \overline{A} \cdot \overline{B} \cdot C + a_2 \cdot \overline{A} \cdot B \cdot \overline{C} \\ & + a_3 \cdot \overline{A} \cdot B \cdot C + a_4 \cdot A \cdot \overline{B} \cdot \overline{C} + a_5 \cdot A \cdot \overline{B} \cdot C \\ & + a_6 \cdot A \cdot B \cdot \overline{C} + a_7 \cdot A \cdot B \cdot C \end{aligned} \tag{2-1}$$

其中，f 表示一个三变量布尔函数；A、B、C 为三个输入变量；“·”和“+”分别表示逻辑“与”和“或”运算。式中 $a_i \in \{0,1\}$，$0 \leqslant i \leqslant 7$，代表函数的真值表信息。式(2-1)是函数的最

小项规范展开式，这是因为每一个由“与”运算组成的乘积项都包含所有的输入变量，即它们都是最小项。一个 n 变量的逻辑函数共有 2^n 个最小项。考虑到 n 位地址线的存储器共有 2^n 个存储单元，每个单元可以存储多位的二进制数据。因此可以直接利用相对比较成熟的可编程只读存储器 PROM 技术来实现逻辑函数。只要改变存储器所存储的数据，就可以达到实现不同的可编程逻辑函数的目的，这就是可编程逻辑函数最简单的实现方式。

图 2-3 为基于 PROM 结构来实现可编程逻辑函数的结构示意图。它主要由上半部分的地址译码器和下半部分的存储体构成，两者均由可编程的熔丝开关和二极管开关阵列组成。图中 A 和 B 为两个输入变量，f_1 和 f_2 为两个输出函数。一般早期 PROM 的可编程开关是利用熔丝技术来实现，即在默认状态下，熔丝开关处于“导通”状态。如果需要开关处于“断开”状态，那么利用编程器把熔丝烧断即可，但是烧断的熔丝不能再连通，因此 PROM 器件只能够实现单次可编程。在图 2-3 的地址译码器中，所有的熔丝都是导通的，这样就可以译码产生所有的地址。当熔丝导通时，二极管的开关状态就由它两端的电压降控制。二极管两端加正向电压则它处于“导通”状态，反之处于“截止”状态。两个输入变量经过反相器后，从上到下输出值为 $\overline{A}$、A、$\overline{B}$ 和 B。以左上角标记为“1”号的二极管为例，如果输入 A 为 1，那么 A 的反相器输出后为 0。这样“1”号二极管一端通过电阻接到电源高电平，另一端是低电平“0”，因此两端就存在电压降，该二极管处于导通状态，导通电流在电阻上存在压降，对应的地址译码器输出，即图中最左边的竖直连线输出就为低电平 0。同理，输入 B 为 1 时，最左边的竖直连线输出就为低电平 0。由于该输出是由一个上拉电阻接到电源，因此只要“1”号二极管和“2”号二极管有一个导通，那么这个输出就为 0。换句话说，只要输入 A 或者输入 B 为 1，那么最左边的竖直连线输出就为 0。另外，如果输入 A 为 0，那么 A 的反相器输出后为 1，这样“1”号二极管两端的电压都是“1”，因此二极管截止，它就不会产生导通

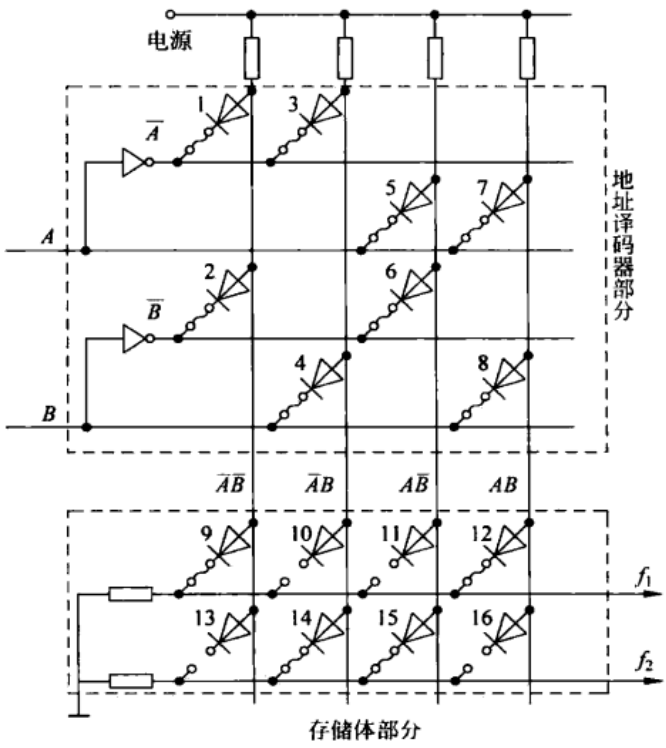


图 2-3　基于 PROM 的 PLD 逻辑函数实现示意图

电流。但是此时的输出不一定为1。只有此时输入 B 也为0，“2”号二极管也截止，输出才为1。由此可见，只有当输入 A 和 B 均为0时，地址译码器中最左边的竖直连线输出才为1，即对应的函数为 $\overline{A}\overline{B}$。依此类推，地址译码器的4个输出从左到右分别为：$\overline{A}\overline{B}$、$\overline{A}B$、$A\overline{B}$ 和 AB。从存储器的角度来说，这4个输出分别对应于二输入地址线不同的地址值。从逻辑函数的角度来说，这4个输出就产生了4个最小项。因此，这里的地址译码器就相当于最小项发生器；地址译码器也相当于图2-1中的可编程与阵列，只不过它把所有的最小项都产生出来了，不管后面的可编程或阵列是否会使用。

再来看图2-3中下半部分的存储体部分。从存储器的角度来看，地址译码器产生了所有的地址值，存储体部分只需要存放对应地址的数据值即可。以图中的函数输出 f_1 为例，如果对应于某个地址的数据为1，那么就让相应的熔丝开关处于连接状态，使得地址线输出的高电平输出经过二极管，在电阻上产生压降，这样 f_1 输出就为1。如果对应于某个地址的数据为0，那么就让相应的熔丝开关处于断开状态，这样就无法把地址线的高电平输出连接过来，f_1 所连接的电阻上就没有电流，因此输出为低电平0。所以存储体的编程操作就很简单，只要让存“0”数据对应的地址线所连接的熔丝处于断开状态就可；存“1”数据的地址线所连接的熔丝处于默认的连接状态。对于输出 f_1 或者 f_2 来说，只要在存储体中它所连接的熔丝有一个处于导通状态，那么地址译码器输出的高电平就能驱动电阻，使得输出为高电平。因此从逻辑关系上来说，地址译码器的输入和输出 f_1、f_2 之间是逻辑“或”的关系，因此图中的存储体部分也就相当于图2-1中的可编程或阵列。在图2-3中，f_1 和 f_2 各有两个熔丝开关处于断开状态，它们分别连接到标号为“10”、“11”和“13”、“16”的二极管。因此根据不同状态的熔丝所连接的地址线位置，就可以得到如图2-4所示的存储器数据以及对应的逻辑表达式。此时 f_1 和 f_2 分别实现了“异或”和“同或”运算。只要改变存储体中熔丝开关的不同编程状态，就可以很方便地实现其他的逻辑功能。

地址(最小项)	存储值 f_1	存储值 f_2	f_1 的逻辑表达式	f_2 的逻辑表达式
00($\overline{A}\overline{B}$)	1	0	$f_1=\overline{A}\,\overline{B}+AB=A\odot B$	$f_2=\overline{A}B+A\overline{B}=A\oplus B$
01($\overline{A}B$)	0	1		
10($A\overline{B}$)	0	1		
11(AB)	1	0		

图2-4 根据开关状态读取存储器数据和相应的逻辑函数表达式

从图2-3中的连接关系可以看出，输出 f_1 和 f_2 的高电平信号是由电源经过一个上拉电阻和存储体中的二极管得到的；输出低电平信号并没有直接受电源的控制，因此输出 f_1 和 f_2 的电压值不是很规范，且驱动能力和抗干扰性很差。从实用的角度来说，输出信号还要增加电平恢复电路，使得输出信号具有较强的驱动能力和抗干扰性能。另外，从存储器的角度来看，输出数据一般是以字节，即8位输出为单位的。图2-3中只画出了2位的输出。因此，每个逻辑函数只需要存储器输出数据中的一位，这些输出就称为位线。而地址译码器的输出是控制输出数据字节中的所有位，因此地址译码器的输出也称为字线。

图2-3所示的PLD示意图结构十分简单，因此我们把这一类器件称为SPLD(simple programmable logic device，简单可编程逻辑器件)。在基于PROM存储器的SPLD实现中，所有的最小项均由地址线产生，不管后续电路是否用到这些最小项。对于有 n 个输入变量的逻辑函数来说，虽然可以采用二维地址译码器结构，但是 2^n 个地址线输出以及存储体

中对最小项的可编程“或”逻辑就显得非常浪费。因此，对于输入变量数超过 4 的逻辑函数来说，把最小项改为乘积项可以有效地减少可编程与阵列的输出项数和可编程或阵列中可编程开关的个数。这就是说，如果把图 2-3 中的最小项输出改为乘积项输出，那么地址译码器就成为可编程与阵列，存储体就成为可编程或阵列，于是图 2-3 就成为了图 2-1 所示的结构。这种结构就是 20 世纪 70 年代中期出现的在结构上比 PROM 要稍复杂的 PLA (programmable logic array，可编程逻辑阵列)结构，它能够完成多种数字逻辑功能。对于确定的函数，一般都可以根据最小项的不同取值，即函数的真值表进行合并化简，从而减少项数和硬件实现成本。从函数化简的角度来看，如果函数的输入变量数小于 5，就可以利用卡诺图进行手动优化；对于多变量的大规模函数来说，手动优化已经不再可行。美国加州大学 Berkeley 分校的 Espresso 开源学术工具是学术界公认为最好的 PLA 组合逻辑优化工具[2]。我们利用该工具就能够针对基于“与”、“或”、“非”运算的大规模逻辑函数得到很好的优化效果。这种组合逻辑优化算法相对来说十分成熟，它有效促进了早期基于 PLA 结构及其可编程逻辑函数的优化。

在 PLA 结构中，与阵列和或阵列都是现场可编程的。由于软件优化算法在可编程与阵列中能够提供足够的灵活性和优化能力，假设把 PLA 结构中的可编程或阵列改为固定的连接关系，那么电路的实现成本就会降低，并且固定连接要比可编程连接的延时要小，电路速度又能得到提高。这种在 PLA 的基础上进行结构简化和性能改进的 PLD 就称为 PAL (programmable array logic，可编程阵列逻辑)器件。对于 PAL 器件来说，“或”阵列的内部连接关系在芯片制造时就已经确定，这部分电路就不再具有现场可编程性。与 PLA 结构相比，PAL 的优点是结构简单，电路延时小，成本低，但是电路实现的灵活度没有 PLA 高。需要指出的是，成熟的组合电路优化算法[2]已经足够弥补硬件上部分可编程灵活性的缺失。从商业的角度来看，PAL 器件的这些优点使得它比 PLA 器件更为普遍，它所能实现的电路已经远远超过了当时基于 74 系列的 TTL(transistor-transistor logic，晶体管晶体管逻辑)门电路设计规模。以下举例说明基于 PAL 结构的电路实现方法。

例 2-1　图 2-5 为简单的 PAL 结构示意图，其中 A、B、C 为三个输入变量，它们经过三个反相器后分别输出 $\overline{A}$、$\overline{B}$ 和 $\overline{C}$。可编程“与”阵列中连线交叉处的“×”表示水平和竖直连接线已经配置为“连接”方式，否则它们之间是没有连接关系的。因此根据图中的连接关系，就可以知道四个与门的输出从上到下依次为：$AB\overline{C}$、$\overline{A}BC$、$\overline{A}\,\overline{B}$ 和 ABC。经过两个固定连接关系的或门后，输出的两个函数值分别为

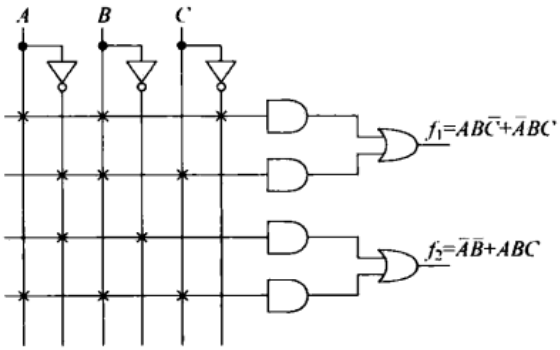


图 2-5　利用 PAL 结构实现数字电路示例

$$f_1 = AB\overline{C} + \overline{A}BC; \quad f_2 = \overline{AB} + ABC$$

对于各种不同的逻辑函数，PLD 提供的开发软件能够自动根据 PLD 硬件结构产生正确的可编程文件。这些文件通过编程接口下载到器件后就配置了所有的可编程开关，从而实现了所需要的逻辑电路功能。

由于早期的 PLA 和 PAL 器件都是基于现场一次性可编程的 PROM 技术，因此无法实现多次可编程或者无法实现较大规模数字电路。20 世纪 80 年代美国 Lattice 公司推出了基于逻辑宏单元的 GAL(通用阵列逻辑)器件，它是基于 EEPROM 技术的中规模器件，实现了电可擦除、电可改写的用户现场可编程功能，集成度比 PAL 要高，可以实现一般的中等规模数字电路。因此 GAL 非常适合于中等电路规模的电路原型设计，在 20 世纪 80 年代得到广泛使用。从逻辑运算的角度来看，GAL 器件的结构基本上还是基于与或乘积项的逻辑运算。关于 GAL 和早期可编程逻辑器件的介绍可参考 Lattice 公司的器件手册或文献[3]，在此不再赘述。

到 20 世纪 80 年代中期，由于半导体工艺技术的进步和对集成电路规模和性能的不断需求，美国的一些集成电路公司分别推出了基于乘积项结构的扩展型 CPLD 器件。简单地说，传统的 CPLD 器件主要是把多个 SPLD 单元及它们之间的层次化互连资源集成到一个芯片中去，使其具有更大的数字信号控制和电平转换能力，非常适合于胶合逻辑应用，但是 CPLD 的数据计算能力明显落后于当时普遍采用的数字微处理系统。20 世纪 80 年代典型的数字微处理器系统包含微处理器、存储器和一些特殊功能的中小规模逻辑器件。为了进一步提高可编程芯片的数据计算能力，1985 年，美国 Xilinx 公司推出了世界上第一款 FPGA 系列产品，包括两个可编程器件 XC2064 和 XC2018 以及支持布局布线的后端设计与编程下载工具。从硬件结构上看，FPGA 器件采用了基于 LUT 的结构，多个 LUT 可以通过互连资源级联起来以实现更为复杂的组合逻辑函数，因此 FPGA 比早期基于与或乘积项结构的可编程逻辑器件具有更强的逻辑函数实现能力。另外，FPGA 芯片内部包含了更多的触发器和复杂的互连资源，可以方便地实现各种复杂的流水线型数字信号处理系统。

从现场可编程逻辑器件的发展历程来看，它们具有以下特点和发展趋势。

(1) 器件具有现场可编程性，即用户在使用芯片的现场，而不只是在芯片设计时才可以改变芯片的功能。和专用集成电路相比，可编程逻辑器件具有通用的逻辑结构和互连结构。不管是基于与或运算的乘积项，或者是基于 LUT 的逻辑单元，以及它们之间的连接方式都是由用户通过销售商提供的软件工具快速地编程。2.2 节将讨论控制这些逻辑单元和互连资源的各种硬件可编程技术。

(2) 硬件可编程依赖于存储器技术。早期的可编程逻辑器件可以直接利用存储器进行设计，包括可编程只读存储器、紫外线可擦除只读存储器和电可擦除只读存储器三种。从外部的电路行为来看，存储器和逻辑函数两者之间是等价的：逻辑函数的输入等价为存储器的地址，函数输出即为存储器的数据。即使对于近期上市的大规模 FPGA 器件，内部的配置信息都是存放于各种存储单元之中，包括 SRAM、Flash 和其他的存储器单元结构。从存储器的意义上来说，冯·诺依曼等人提出的存储程序(stored program)计算的思想同样适用于可编程逻辑技术，即计算和控制都可以作为数据的方式存放于存储器中。只不过对于通用的计算机来说，存储程序中存放的是以字节为单位的粗粒度指令流数据，而在可编程逻

辑器件中存放的主要是以位为单位的细粒度配置数据。这两种类型数据在计算机运行时或者可编程器件工作时一般都是保持不变的。

(3) 可编程逻辑器件内部有两种不同功用的存储单元：只读的可配置存储单元和可读写的用户数据存储单元。例如以最简单的基于 PROM 的 PLD 器件为例，图 2-3 中熔丝代表可编程配置开关，在器件配置后这些开关的状态就一直保持不变，尽管用户输入数据不断变化。这也是 PROM 作为可编程存储器的根本原因。然而二极管的开关状态是在器件运行时随着不同的用户输入而不断改变。从存储器的角度来说，这些二极管状态所对应的数据为动态数据，两个不同的状态分别对应于用户数据中的“0”和“1”值。这种区分非常类似于通用计算机中的只读可执行程序和用户数据之间的差别。用户代码经过编译后就产生可执行程序，编译完成后可执行程序就不再改变。在程序运行时，存储器中的用户数据不断变化。由于细粒度可编程逻辑器件中的配置数据相对于指令流数据比较容易控制，因此近期在学术领域就诞生了在可编程逻辑器件运行时能够动态修改自身配置信息的动态可重构系统。这部分内容将在第 6 章介绍。

(4) 可编程逻辑器件继续保持着由控制逻辑到计算电路的发展趋势。由于它们的输入输出电平非常灵活，并且内部的可编程逻辑单元很适合于控制逻辑实现，因此可编程逻辑器件早期普遍作为胶合逻辑使用。随着 20 世纪末 FPGA 的普及和系统级 FPGA 器件的出现，可编程逻辑具有数据宽度可调的嵌入式 DSP 单元、存储器硬核和嵌入式微处理器核等粗粒度计算单元。系统级 FPGA 除了传统的控制逻辑外，在粗粒度计算方面的功能和性能有了显著的提高。这就是 2.1.2 节要讨论的“数据通路与 FPGA”内容。

2.1.2　数据通路与 FPGA

实现复杂的数据计算是通用处理器和数字电路与系统中比较耗时的部分，当然这些数据计算是需要相应的控制电路来配合的。在通用型微处理器、DSP 处理器或者一般稍微复杂的数字电路系统中，数据通路是指用户数据在存储器、寄存器和各种算术或逻辑计算单元之间的传送路径，包括常见的乘法器、加法器、移位、与或非、异或等算术逻辑单元。数据通路是在控制电路的操作下完成用户所需要的各种数据运算。对于通用的微处理器来说，存放于主存储器中的用户数据经过高速缓冲器(cache)被送到寄存器组，ALU 单元就对寄存器中的数据进行逻辑或算术操作，计算后的结果通过寄存器、缓存器返回到主存储器。数据通路传统上包括挂在数据总线上的各种计算和存储单元。而数据传送流程需要复杂的控制逻辑支持。

基于乘积项的可编程逻辑器件主要是基于“与”和“或”逻辑操作，它非常适合于逻辑控制应用，但对算术运算的实现效率较低，或者说它们不适合于复杂的数据计算。从这个意义上说，用与或非等逻辑操作来实现算术运算不是很有效的途径。例如异或门实际上是一位半加器，但是它所需要的晶体管数目比与或非门要多，面积要大。因此基于乘积项的可编程逻辑单元不适合于实现快速高效的数据通路中的各种计算。另一方面，从通用型微处理器的结构来说，数据通路主要用于在内存、缓存和寄存器组等各种存储单元之间实现数据传递，ALU 单元用于计算所占的时间比例很少。虽然计算机结构中的总线标准、速度和带宽不断提高，但是数据通路中存储器和 ALU 之间的数据通信依然是通用计算效率的瓶颈所在。因此，如何突破传统可编程器件中的基于逻辑“与”和“或”的乘积项结构，同时减少通用

处理器中存储器和计算单元之间的通信成本是提高可编程器件数据通路计算效率的关键。下面我们就从通用处理器的数据通路结构出发来突破基于可编程逻辑“与”和“或”的乘积项结构，从而更好地理解当前已有的系统级 FPGA 结构及其发展趋势。

与通用 CPU 相比，FPGA 使得硬件电路具有可编程和可重构的能力，用户可以根据不同的需求很方便地利用 FPGA 来构建电路。但是与 CPU 的通用性不同，FPGA 的可编程性是细粒度的。相比于提高总线速度和位宽的解决方法，一种降低通用处理器数据通路中 ALU 和各种存储单元通信成本的办法就是拆分传统的总线结构，即先把总线拆为独立的互连线，如图 2-6 所示。失去总线接口的 ALU 就被拆分为细粒度通用计算单元，例如 LUT 单元。一个 LUT 单元就是一张查找表，它能输出对应于输入地址所存储的一位数据，因此它可以作为细粒度的存储单元。另外，LUT 还可以用于实现布尔函数或者是通用的逻辑计算函数，因为一个细粒度的 LUT 只有一位输出，所以多位数据可以由多个 LUT 组合输出得到。如果这些 LUT 能够被紧密地放在一起，就能够实现以“字节”为单位的数据输出。可以说，一个 ALU 单元可以用一组基于 LUT 和触发器的可编程逻辑单元来代替。这些单元既可以作为分布式存储，也可以实现细粒度和粗粒度的计算。另外，传统微处理器结构中和总线相连的存储单元就用细粒度的触发器或彼此独立的寄存器代替。这些触发器或寄存器的数据输入端可以直接连到基于 LUT 的可编程逻辑单元输出端，这样它们之间的通信并不需要复杂的时序控制。一般来说 LUT 的输出通过一个简单的多路数据选择器(multiplexer，MUX)就能连到触发器或者直接作为组合逻辑输出，从而级联实现复杂的组合逻辑功能。如图 2-6 所示，为了减小传统处理器中 ALU 和存储单元之间的通信成本，拆分总线后得到的基于 LUT 的 FPGA 阵列单元就能完成各种细粒度的逻辑和算术运算。LUT 和触发器或寄存器通过内部的互连资源连接，相互之间的通信成本很低。这个细粒度单元的各种工作模式由内部的可编程配置开关控制。这些可编程开关也经常称做“码点”。码点的功能就相当于微处理器中指令的作用。不同的配置码点就实现了不同的电路，这就如同在微处理器中，不同的指令执行不同的数据操作一样。在通用微处理器中，指令流基本上是串行执行的，而 FPGA 中的各种可编程码点在运行前就已经配置好了。一个芯片中可以包含成千上万个如图 2-6 所示的细粒度 FPGA 阵列单元，这些单元之间又配备了数十条甚至数百条可编程互连线。相对于总线结构来说，这些互连资源可以并行地实现点到点的连接，从而保证各可编程单元能够进行高效的并行计算。因此，从计算并行性、灵活性和通信成本的角度来看，图 2-6 中的 FPGA 阵列单元是非常适合于通用计算的。相对于通用微处理器来说，FPGA 的主要缺点是使用起来不是很方便。换句话说，编写串行化的高级软件语言代码比编写并行化的硬件描述语言代码要容易很多。当然同专用芯片相比，利用可编程逻辑实现的各种粗粒度计算的效率还是比较差的。为了进一步提高计算的粒度、便利性

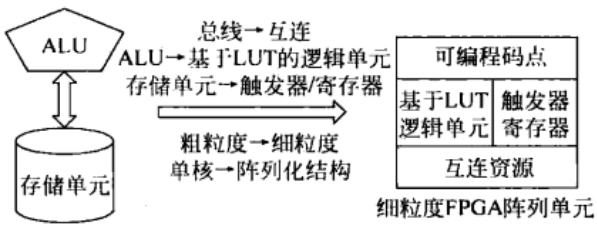
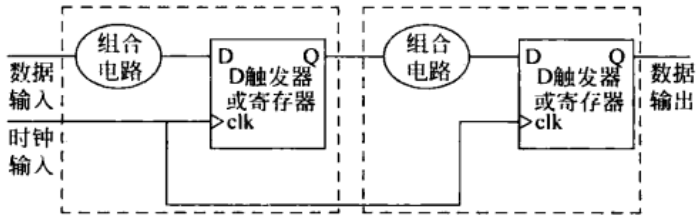


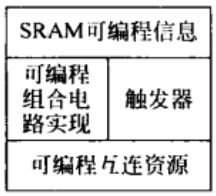
图 2-6 从通用微处理器到 FPGA 阵列单元结构

和并行性，系统级 FPGA 一般都直接嵌入了多个粗粒度的计算单元，例如存储器、乘法器或者 DSP 单元以及嵌入式微处理器等。

相对于通用处理器或者 DSP 处理器来说，图 2-7(a)是典型的 RTL 数字电路结构。这种结构非常适合于 FPGA 实现，这也是 FPGA 一开始普遍用于专用集成电路的原型验证的主要原因。下面从数字电路结构的角度来讨论图 2-6 所示的 FPGA 阵列单元结构的优越性。



(a) 典型的RTL电路基本结构



(b) FPGA可编程逻辑单元基本结构

图 2-7　RTL 电路单元和 FPGA 可编程逻辑单元结构的对应关系

如图 2-7(b)所示，在 FPGA 可编程逻辑阵列单元中，可编程码点所控制的一个逻辑函数的输出、一位触发器(当数据为多位时就是寄存器)存储和一根互连的连接，它们不再像 CPU 结构中以字节为单位的指令、数据和总线结构。从目前商用多核芯片的规模来看，一个多核芯片能容纳的 CPU 核数目一般小于 1000 个。但是一块 FPGA 芯片能包含成千上万个可编程逻辑阵列单元，这些可编程逻辑单元可以并行工作。在主流的 FPGA 器件中，组合逻辑电路部分主要是通过 LUT 来实现的。一个 n 输入的 LUT 能够实现任意的 n 输入函数。多个 LUT 的级联与并行能实现更为复杂的逻辑函数。因此，如果一个应用能够在 FPGA 上实现，那么 FPGA 实现一般要比多核 CPU 程序具有更高的并行计算效率。

对于典型的 RTL 电路单元，如图 2-7(a)中的虚线框所示，它包含了组合电路和 D 触发器或寄存器。比较图 2-7(a)中的虚线框、图 2-6 中的 FPGA 阵列单元结构和图 2-7(b)中的 FPGA 可编程单元结构，可以发现“可编程组合电路实现”一般是基于 LUT 的结构，其函数功能由 SRAM 可编程信息决定。另外，SRAM 可编程信息也决定了可编程组合电路和触发器之间的连接方式。为了满足可编程单元的灵活性并提高芯片面积的利用率，可编程逻辑电路和触发器一般有多种连接方式。图 2-7(b)中的可编程互连资源用于多个可编程单元之间的连接，从而构建更大的电路结构。如图 2-19 所示，多个可编程逻辑单元通过互连资源构成了更大的组合和时序电路。经过 20 多年的发展，源于 FPGA 的可编程逻辑器件已经适合于较为复杂的数据通路计算。可编程逻辑器件应用领域从早期的胶合逻辑，到算法逻辑设计、专用芯片的原型验证和数字信号处理以及嵌入式系统领域，获得了广泛的应用。

2.2 常用的硬件可编程技术

如 1.2 节所述，可编程性主要是指能提供给用户一种底层的硬件编程机制，使它具有功能定制的属性。只要提供给用户基于通用处理器硬件环境和相应的软件编程工具，用户就可以利用计算机语言通过指令的方式进行编程。这种可编程的能力可以理解为软件可编程性，它在底层硬件方面需要以指令集的电路实现作为前提。而硬件可编程性主要是提供给用户一种硬件电路设计环境，用户可以利用相应的编程工具来设计具体的数字电路。不管是软件可编程性或者是硬件可编程性，它们都无法脱离存储器技术。这些存储器用于保存对应于软件可编程性的可执行程序和对应于硬件可编程性的配置数据。

2.2.1 配置数据和用户数据的区别

20 世纪 70 年代的硬件可编程电路就是指用户可以一次性或者多次现场可编程的各种逻辑器件，例如 PROM、EPROM、EEPROM 等器件。虽然从名称上来说，FPGA 被称为现场可编程门阵列，但是现场可编程性并不局限于 FPGA。只要一个器件能够提供一定的硬件可编程能力，即用户在使用的现场，在不修改硬件运行环境的条件下就能实现不同功能，那么这种器件就是一种现场可编程器件。用户使用随器件配套的软件工具把定制的程序或者用户数据烧写到可编程逻辑器件。对于通用处理器，软件可编程性就是把可编程信息，包括指令、只读数据和用户可读写数据等，写到各种存储器。这些可编程信息用于配置处理器的功能并向计算单元提供输入数据。用户编写的高级语言代码经过编译后放到非挥发的硬盘存储器；在运行时这些数据一般在操作系统的调度下放到内存。与非挥发的外部存储器相比，内存一般是指需要刷新的动态存储器(DRAM)。通用微处理器在执行时根据需要把这些指令流和数据流由内存搬到高速缓冲器，其中的指令流用于处理器功能的配置，而数据流则用于进行数据通路计算。高速缓冲器一般使用静态存储器(SRAM)单元结构，以保证数据的传输速度。从硬件成本来说，单个 SRAM 单元所需要的晶体管数量要比 DRAM 单元多。对于复杂的处理器来说，一般还需要多级缓存。计算后的结果也都要通过多种存储器输出到主内存。对于大量的数据计算来说，一种提高计算效率和数据吞吐率的办法是让指令流和数据流通过各自的总线来访问存储器。这种结构便是常用于 DSP 处理器的哈佛结构。

对于 FPGA 来说，它和处理器不同，运行时一般不是通过指令的方式进行操作。FPGA 所实现的电路功能需要事先配置下载，没有配置前的 FPGA 器件就如一张白纸，不具有任何电路功能。我们可以把这种用于定制 FPGA 功能的二进制数据称为“配置流”[4]。和用户数据相比，配置流数据在 FPGA 运行时一般是不可变的，就好像通用处理器运行时它的可执行程序一般是只读方式一样。如果 FPGA 在运行时能动态更新电路本身的功能，那么这个系统就是可重构系统了，可重构系统的详细内容参见第 6 章。下面通过 LUT 和 MUX 两者的比较来理解配置信息和用户数据的差别。

图 2-8 是最简单的二选一 MUX 逻辑单元及其晶体管实现电路。它一共有 3 个输入端，1 个输出端。其基本逻辑功能是：当选择端 s 为 0 时，输出端 f 就是输入 x_0 的值；当选择端 s 为 1 时，输出端 f 就是输入 x_1 的值。用布尔表达式描述如下：

$$f = \bar{s}x_0 + sx_1 \tag{2-2}$$

图 2-8 也给出了利用 MOS 传输晶体管实现二选一 MUX 的单元电路，其中反相器只需要两个晶体管就能实现。为了便于和 LUT 比较，图 2-9(a)画出了八选一的 MUX 符号及其输入输出端口。它一共有 11 个输入端，其中 3 个输入端 s_0、s_1、s_2 为选择端，由它们选择其余的 8 个输入端 x_i 中的一个连到输出 $f(0 \leqslant i \leqslant 7)$。用布尔表达式描述如下：

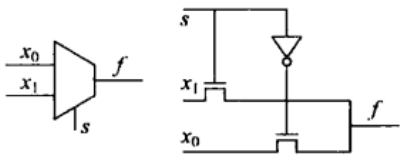
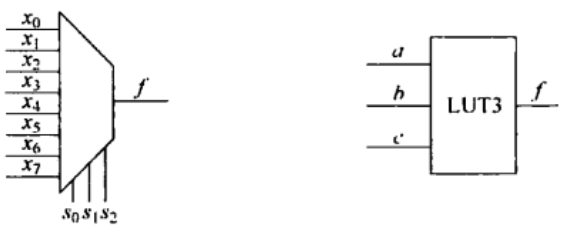


图 2-8　最简单的二选一 MUX 端口及其晶体管实现电路

$$f = \sum_{i=0}^{7} \dot{s}_2\ \dot{s}_1\ \dot{s}_0\ \dot{x}_i \tag{2-3}$$

其中，$\sum$ 表示连续的逻辑或运算；$\dot{s}_j = \begin{cases} \bar{s}_j, i_j = 0 \\ s_j, i_j = 1 \end{cases} (0 \leqslant j \leqslant 2)$，$i_j$ 为 i 的第 j 位二进制表达式的值。例如当 $i = 5 = (101)_2$ 时，$\dot{s}_2\ \dot{s}_1\ \dot{s}_0 = s_2\ \bar{s}_1 s_0$，即 MUX 输出端 f 的值是 x_5。

图 2-9(b)是 LUT3 的符号和输入输出端口。它能够实现任意的三输入单输出函数，只需要把这个函数的真值表配置到 LUT3 内部就可。三输入的 LUT3 内部有 8 个配置数据，用于表达任意的逻辑函数真值表。这些配置数据有时简称为“配置点”或“码点”。比较图 2-9 中的两个图，就可以发现如果把 MUX 左边的 8 个输入端作为配置点使用，即在电路工作时这 8 个值是不变的，那么 MUX 在逻辑上就作为一个 LUT3 使用了。MUX 的三个选择端 s_2、s_1、s_0 就等价于 LUT3 的三个输入端 a、b、c。或者倒过来说，如果把 LUT3 中的 8 个配置点都作为用户输入数据使用，即在电路工作时这 8 个数据值都是可以由用户随时改变，那么它就成了一个 MUX。这就是配置数据和用户数据的差别：前者是静态的，电路工作时值保持不变；后者是动态的，电路工作时值是根据用户输入而变化的。在本节中所讨论的硬件可编程技术，就是指用于存放配置数据的存储单元，而不是用于存放用户数据的存储单元。前者要比后者简单很多。用户数据一般是放在触发器或者寄存器中，它们需要同步时钟及其边沿触发功能，以确保电路稳定快速地工作。配置数据储存单元则相对简单，所需要的晶体管个数少，一般用 SRAM 或者其他结构就可以了。如图 2-6 中的“可编程码点”就是利用 SRAM 单元存储的配置信息。以下首先介绍基于配置数据的硬件可编程技术，然后讨论用于存放配置数据的存储单元结构和基本原理。



(a) 八选一MUX符号和输入输出端口　(b) LUT3的符号和输入输出端口

图 2-9　MUX 和 LUT 的比较

利用不同存储结构的硬件可编程技术及其所控制的基本可编程单元可以构成不同的 FPGA 器件架构。硬件可编程技术可以基于 EEPROM、熔丝、反熔丝、Flash、SRAM 等存储单元，这些存储单元所存放的数据用于控制可编程逻辑单元的函数功能、可编程互连单元的

连接关系和可编程 IO 的输入输出特性。例如早期的可编程逻辑器件是基于 EEPROM 或者反熔丝的与或逻辑阵列结构，互连资源相对简单。20 世纪 80 年代后 Xilinx 公司逐渐推出了基于 SRAM 存储的硬件可编程技术、基于 LUT 的可编程逻辑单元和层次化的可编程互连技术的可编程逻辑器件。还有其他一些公司，例如 Actel、Altera、Anachip、Atmel、Cypress、Lattice、Leopard Logic、Quick Logic、STMicroelectronics、Triscend 等分别采用了不同的可编程技术、可编程逻辑和互连结构的组合，形成了多种多样的可编程逻辑器件[5]。由于可编程器件激烈的竞争环境，以上所列的大部分公司已经退出可编程逻辑器件行业，或者被其他公司收购。本节不讨论各个公司的可编程逻辑结构和它们的优缺点，而是侧重于理解适用于不同硬件可编程技术的各种存储器单元结构及其它们的控制关系。常用存储器的层次结构可以参考第 1 章的图 1-15。由于用于硬件可编程的存储单元只用于配置，而不会用于用户输入数据，在电路运行时这些存储单元信息的值一般是保持不变的，因此存储硬件可编程配置信息的主流结构单元一般采用图 1-15 中顶层的 SRAM 结构，便于提高晶体管的利用率。虽然 DRAM 存储单元要比 SRAM 简单，但是 DRAM 需要刷新，因此不适合于配置数据的存储。如图 2-10 所示，这些配置信息主要控制三方面信息。

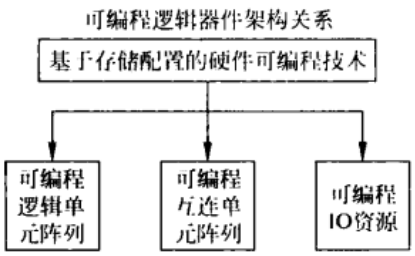


图 2-10 基于硬件可编程技术的基本可编程单元架构关系

(1) 可编程逻辑单元阵列的功能。电路中的逻辑函数主要由这些可编程逻辑单元组合来实现，不同的配置信息就会产生不同的逻辑函数。主流的可编程逻辑单元都是基于 LUT 的结构，具体介绍见 2.3.1 节。

(2) 可编程互连单元阵列。逻辑单元的输入输出可以级联，从而组合成更大的组合逻辑函数和时序电路。因此逻辑单元之间的连接关系就由这些可编程互连单元来实现，而这些互连单元本身是由配置信息控制的。互连结构的具体介绍见 2.3.2 节。

(3) 可编程 IO 资源。可编程器件与片外电路的接口就通过这些可编程 IO 资源来实现。这些 IO 往往需要支持多种 IO 标准，例如经典的 LVTTL(low voltage TTL，低电压晶体管晶体管逻辑)、LVCMOS(low voltage CMOS，低电压 CMOS)、LVDS(low voltage differential signal，低电压差分信号)和 PCI(peripheral component interconnect)总线等接口。具体介绍见 2.3.3 节。

随着半导体技术的不断发展，一个 SRAM 所存储配置信息的电荷量不断下降。并且由于它是电平触发电路，而不是边沿触发，因此电路的稳定性和可靠性受到很大影响。近期学术界有很多针对基于 SRAM 的 FPGA 软错误建模和提高电路稳定性的优化技术研究成果[6]。

这里针对配置数据和用户数据的区别，有必要说明 MUX 单元的灵活性。如图 2-9 中的八选一 MUX 逻辑单元，它可以实现比较复杂的 11 个输入单输出的逻辑函数。如果把它的 8 个输入数据作为配置值固定，其余 3 个输入作为用户输入变量，那么它就成为一个 LUT3，即三输入 LUT，可以实现任意的三输入函数。如果把它的 8 个输入端连到互连线，其余 3 个输入作为用户控制端，那么它就能动态地实现不同的互连方式。因此 MUX 可以用于实现逻辑函数，退化成 LUT，也可以用于可编程互连结构。在商用器件中，一般 MUX

的功能模式是确定的，即它或者用于实现逻辑函数，或者用于控制互连连接。学术界最近提出 MUX 资源可以根据需要既可以作逻辑函数实现，又可以用于互连资源连接[7]，这种灵活性在理论上为提高资源利用率和电路的性能提供了不错的途径。

2.2.2 基于存储的配置技术

最简单的硬件可编程技术就是基于熔丝单元的配置方式，它和日常生活中照明用电的保险丝原理类似。熔丝的默认状态是导通的，如果有大电流通过，熔丝就会被烧断，那么它就处于截止状态，两端所连接的导线就彼此断开。因此配置信息就可以用这两种开关状态来实现。如果它所控制的连线能接到二极管或者三极管等开关元件，那么熔丝结构就可以用于配置二极管、三极管或 MOS 管的状态。图 2-3 中的 PROM 就采用了熔丝状态配置二极管的硬件可编程技术，它的基本结构单元如图 2-11 所示。默认情况下熔丝处于导通状态，因此二极管、三极管或 MOS 管加正向偏置时就导通，加反向偏置时就截止。但是当熔丝被烧断后，不管二极管、三极管或 MOS 管处于何种偏置，它们的状态都不能改变。因此，熔丝的状态和它所控制的二极管、三极管或者 MOS 管的状态不是对等的。熔丝的状态属于配置态，当电路运行时它的状态一般保持不变。图中二极管、三极管或者 MOS 管的状态只有当对应的熔丝开关导通时，才用于实现用户所需要的状态，可以是“导通”或者“截止”。它们对应于用户输入的数据状态。这两种状态的区别有点类似于通用处理器中指令和用户数据的关系，即前者是只读的，后者是可读可写的。

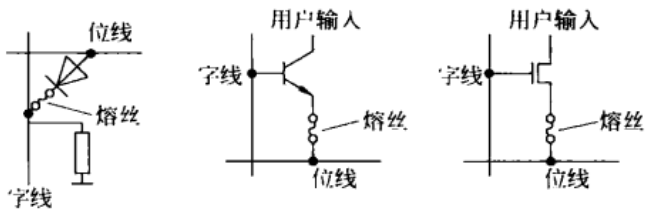


图 2-11 基于熔丝技术的硬件可编程单元：二极管、三极管和 MOS 管开关控制

用于硬件可编程的可配置存储单元很多，除了经典的熔丝结构外，常用的还有反熔丝结构。它和熔丝的差别是默认状态为“截止”。只有当它编程后，状态变为“导通”。由于熔丝和反熔丝都是一次可编程的，即它们的状态只能改变一次，一旦熔丝烧断后，它就不可能再回到“导通”状态。为了实现多次可编程的特点，后来成功地研制了高电压编程和紫外线擦洗的 EPROM 技术、电擦除的浮栅型 EEPROM 技术，以及基于 Flash 的闪存技术。由于这些配置存储单元和标准的 CMOS 工艺不兼容，因此不能充分发挥最先进的半导体工艺技术的优势。主流的大容量 FPGA 器件均采用基于 SRAM 的可编程技术。由于它和标准的 CMOS 工艺兼容，因此可以利用最先进的标准 CMOS 工艺进行制造。而其他的硬件可编程技术一般都要比最先进的 CMOS 工艺起码落后一至二代。

主流 FPGA 的配置单元都是基于 SRAM 的结构，图 2-12 是由 6 个晶体管组成的标准 SRAM 单元电路，其中 M1、M2、M3、M4 分别组成了两个反相器。通过第 1 章的图 1-16 进行比较就能发现两个反相器的位置。两个反相器的首尾对接就构成了一个互锁结构，能够把“0”或者“1”信息锁存住。M5 和 M6 这两个晶体管主要是用于读写配置信息的，用于改变或者读出内部的配置值。当字线为高电平时，M5 和 M6 就导通，两个互补的位线上的值

C 和 $\overline{C}$ 就会写入到内部互锁的反相器单元，或者内部的配置信息就输出到位线。同触发器相比，SRAM 所需要的晶体管数量要少很多。最简单的 DFF 边沿触发器，或者叫做维持-阻塞边沿 D 触发器一般需要 6 个与非门，而每个与非门都需要 4 个 MOS 管，因此用于配置的 SRAM 单元比用于用户数据的触发器要简单很多。这主要是由于配置信息不需要随时改变，而用户数据可能随时变化。这也说明简单的 SRAM 单元要比第 1 章图 1-15 所示层次结构中的下一级 DFF 单元在工作稳定性方面有着先天的劣势。

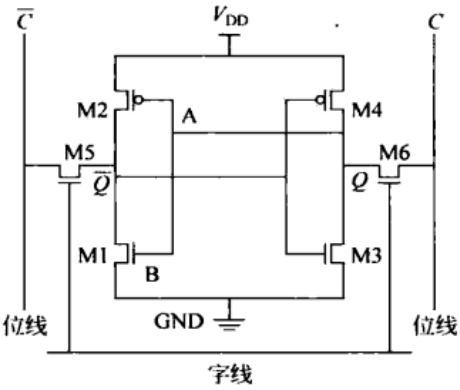


图 2-12 标准 6 管 SRAM 单元电路

图 2-12 中的 SRAM 单元在配置电路中一般简化为“SRAM 配置点”，如图 2-13 所示。图中画出了基于 SRAM 单元的最常见的三种配置电路。第一种是利用 SRAM 配置值控制传输管的导通和截止状态。如果该传输管的源端和漏端分别接到两根互连线，那么这个传输管就用于控制互连方式。如果该传输管用于实现 LUT 内部的电路，那么它就用于实现可编程单元内部的逻辑函数。这是一种最简单的可编程实现方式，只需要一个配置点和一个晶体管即可，并且能够实现双向连接。但是由于传输管存在阈值损失，因此它的驱动性能不是很好。第二种方式是利用配置点选择不同的互连连接方式。图中画出了两个配置点控制四根输入线的连接。当这两个配置点取不同的值时，四根输入线中就被选择不同的连线连到输出端。这种配置方式只能控制输入到输出的单向连接。第三种配置方式是利用一个配置点控制三态缓冲器。当配置值为高电平时，三态缓冲器导通；否则三态缓冲器截止，输入到输出的连接就会断开。

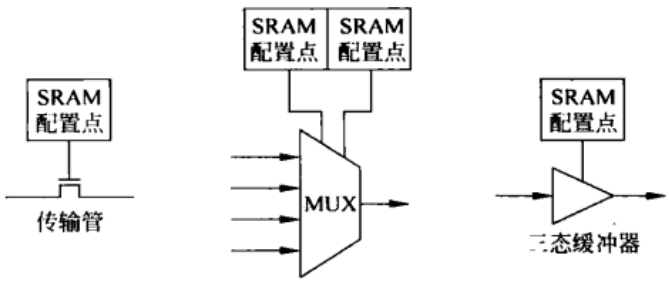


图 2-13 基于 SRAM 单元的常见配置电路单元[8]

目前在主流的 FPGA 器件中，由于 SRAM 的制造工艺与 CMOS 标准工艺兼容，因此它是最常见的配置方式。表 2-1 对常用的硬件可编程技术进行了简单的对比[10, 11]。

由于经典的 PLD、CPLD 或者 FPGA 器件都是基于细粒度的可编程逻辑单元，不管是与或非逻辑运算还是 LUT 单元，基本输出函数都是单变量的。这些单元对于实现以字节为单位的数据运算就需要较多的可编程逻辑单元，效果不太理想。因此近十年来由于系统级 FPGA 的需求，在传统的可编程逻辑器件基本架构上添加了许多粗粒度的单元。这里的粗粒度是相对以“位”为单位的逻辑函数来说，能够直接以“字节”为基本单位进行数据计算和操作的嵌入式微处理器、嵌入式存储器和乘加单元等。它们的功能配置还是通过类似于 SRAM 的单元进行实现，具体介绍见 2.4 节。

表 2-1 常用硬件可编程配置存储单元特性比较[10, 11]

硬件可编程技术及性能	SRAM	反 熔 丝	EEPROM 和 Flash
制造工艺兼容性	与标准 CMOS 工艺兼容	不兼容	不兼容
多次可编程性	支持在线多次可编程	一次可编程	支持在线或离线多次可编程
重复可编程速度	快	没有重复可编程功能	比 SRAM 慢 3 倍以上
易失性	是(掉电后数据丢失)	否	否
可编程配置单元尺寸	大(6 个晶体管的标准单元)	很小(不需要晶体管)	小(1～2 个晶体管)
开关电阻	约 500～1000Ω	约 20～100Ω	约 500～1000Ω
开关电容	约 1～2fF	＜ 1fF	约 1～2fF
功耗	中等	小	中等
安全性	尚可	安全	安全

除了利用 SRAM 等单元进行细粒度和粗粒度单元的功能配置外，近几年也出现了针对电路性能的硬件配置技术。主要表现在以下方面。

(1) 出于低功耗设计的需求，我们可以利用 SRAM 单元实现可编程功耗或可编程电源管理技术。例如对于电路中关键路径上的逻辑单元采用高电压供电，保证其工作频率；而在非关键路径上的大部分可编程逻辑资源采用低电压工作方式，从而降低了功耗；对于芯片中没有用到的可编程逻辑资源，则可以采取断电的方式，进一步降低芯片消耗的功耗。因为不同电路的关键路径位置、资源利用率等因素即使在同一个芯片上运行都是不同的，所以这些低功耗设计技术就需要配置数据进行现场控制。

(2) 为了减少芯片输入输出的延时和匹配，可编程逻辑器件的输入输出一般都支持多种驱动能力和延时匹配的配置功能。另外芯片的输入输出还支持多种电平标准或通信协议标准，这样可以显著扩展芯片的应用领域并提高电路的性能。

(3) 为了提高芯片中时钟的同步和速度问题，FPGA 器件一般都支持嵌入式的锁相环 IP 硬核。它们可以对输入的参考时钟进行倍频、分频和相位调整。这些嵌入式锁相环的性能要比片外时钟发生器的性能优越很多。锁相环的不同工作模式选择也都需要配置信息进行控制。

随着半导体工艺的发展，我们相信硬件可编程技术在电路的功能、性能、功耗和可靠性方面将发挥更大的作用。

2.3 经典 FPGA 的硬件结构

在介绍了硬件可编程技术后，本节基于图 2-10 继续讨论经典 FPGA 的硬件结构。从 2.2 节所讨论的硬件可编程的角度出发，将经典 FPGA 按照功能分为可编程逻辑单元、可编程互连单元和可编程 IO 这三个基本组成部分，我们分别用三个小节进行阐述。对于系统级 FPGA 中的一些可配置粗粒度单元结构，将在 2.4 节讨论。

2.3.1 可编程逻辑单元

对于可编程逻辑单元至今尚没有明确一致的定义，不同的公司和论文经常采用不同的

名称，例如 slice、CLB(configurable logic block，可配置逻辑块)、LC(logic cell，逻辑单元)或者 BLE(basic logic element，基本逻辑单元)等。有时候它表示只包含一个组合电路输出和一个时序电路输出的基本单元，有时它也用于描述由这些基本单元打包在一起所组成的可编程逻辑簇[8]。本节主要讨论由一个查找表和 DFF 构成的基本可编程逻辑单元结构。有兴趣的读者可以阅读参考文献[8]的第 6 章来理解逻辑簇的结构特点。作为通用型的基本可编程逻辑单元应该满足以下特点。

(1) 完备性：可编程逻辑单元能够实现适度规模的任意组合函数。如果逻辑函数规模超过一个基本单元的容量，那么可以通过多个逻辑单元的级联来实现，这是作为可编程逻辑单元的最基本要求。一个逻辑单元所能实现的函数个数越多，它的完备性和灵活性就越好。

(2) 可编程性：用户可以通过配置相应的编程点信息，实现具体的功能函数。这是现场可编程性的基本需求，即用户不需要另外流片就能实现他所需要的功能。从结构上来说，逻辑单元中必须存在一些可配置的存储单元，用于定制用户的电路。这些配置单元必须和用户电路中所需要的触发器或者寄存器等用户数据存储单元在硬件结构上彼此独立。这些配置值在电路工作时一般是保持不变的。

这里我们假设现场可编程性是指默认的静态可编程。与之相对应的动态可编程性是指电路在运行时改变本身的配置值。同样以图 2-9 中的 MUX 和 LUT 为例。图中的 MUX 并没有可编程性，这是因为 MUX 单元不具备可配置存储单元，它的所有输入和输出都是给用户电路用的。但是图中的 LUT 就具有现场可编程性，这是因为 LUT 只有 3 个输入，它内部需要 8 个配置存储单元。用户可以利用这些配置存储单元进行定制功能，设计任意的三变量函数。虽然从逻辑意义上来说，图中 MUX 的输入数远远多于 LUT 的输入数，它所能实现的函数个数也远远超过了 LUT 所能实现的函数个数，但是单纯的 MUX 本身并不具有现场可编程性。因此从现场可编程性的角度来衡量，主流的 FPGA 器件均以 LUT 作为可编程逻辑单元的核心结构。

(3) 可扩展性：除了完备性和可编程性以外，基本可编程逻辑单元应该具有灵活的接口或连接方式，以便实现更大规模的组合电路和时序电路。例如为了提高加法器的使用效率和性能，可编程逻辑单元中一般都包含了快速进位链的电路单元。这样相邻的可编程逻辑单元可以级联，快速生成进位逻辑。虽然不包含进位电路的可编程逻辑单元也能够实现多位加法器，但是电路性能要差很多。因此，可扩展性是在完备性和可编程性的基础上，对电路性能的基本需求。

从完备性的角度来看，众所周知“与非门”是一种通用的逻辑门，利用与非门和锁存器或者触发器就可以实现任意的逻辑函数(锁存器或触发器本身也可以通过与非门搭建起来)。从最极端的角度来说，任何数字集成电路都是由晶体管实现的。因此我们当然可以把晶体管作为最小的可编程开关单元。可以猜测这种结构在晶体管的互连上将耗费大量的可编程开关，因此面积、速度和功耗性能都不够好。20 世纪 80 年代开始流行的门海(sea of gates)结构，或者更准确地说是晶体管阵列结构就是以晶体管作为基本单元来构建用户电路。随着数字电路规模的扩大以及对电路性能的要求不断提高，门海技术已经用得很少了。与非门比晶体管的粒度要大些，但是与非门本身不具有现场可编程性，因此所有的可编程性都同样需要外加的可编程单元及其互连单元来实现。这种基于与非门的可编程逻辑单元同样会面临可编程资源极大浪费的问题，除非所实现的数字电路规模很小。1989 年，英国一家微

电子公司 Pilkington Microelectronics 就推出了基于与非门和锁存器结构的可编程器件[12]。如图 2-14 所示，所推出的 ERA(electrically reconfigurable array)60100 器件中与非门的两个输入端的连接方式均由 MUX 控制，从而提供现场可编程功能。MUX 的数据选择端由配置点控制。例如图中间的与非门 A 有两个输入端，它们经过 MUX 来选择连接关系。和该与非门 A 相邻的五个与非门 B、C、D、E、F 的输出均可以作为它的输入。究竟选择哪个输出作为 A 的输入，需要码点来控制。与非门 A 与 B、C、D、E、F 连接的互连线都是很短的，称为“局部互连线”，如图 2-14 左下角的标记。除了局部互连线外，还有竖直长线、竖直短线和水平长线等互连资源。如图中第一行的互连通道所示，一条水平长线的长度，即它所跨的与非门的数目，要远远大于局部互连线的长度。同理，图中间的竖直短线连到第五行与非门就终止了，而竖直长线还可以继续往上下连接。由于一个与非门所能实现的逻辑函数十分简单，这样就需要多个与非门通过不同长度的互连线来构建规模较大的电路。由于这种由一个与非门组成的可编程单元的逻辑功能过于简单，对于稍微复杂点的电路，就要占用大量的互连资源。例如一个 4 位加法器就要用掉大约 130 个与非门，它不能适应于大电路的应用[12]。因此类似的产品很快就被更具竞争力的可编程逻辑器件所淘汰。

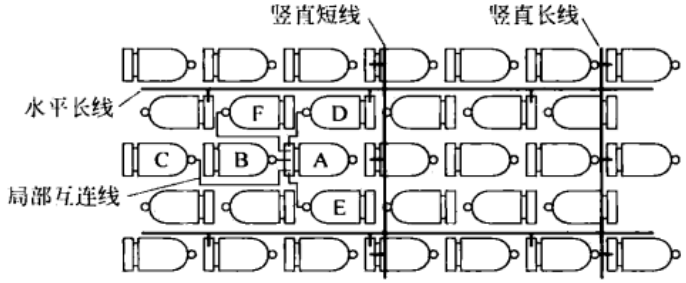


图 2-14　包含 10000 个与非门单元的 ERA60100 可编程逻辑器件结构

为了提高可编程逻辑器件互连资源的利用率和性能，一种可行的办法就是提升可编程单元的粒度，即从单个晶体管，到与非门，或者是图 2-1、图 2-5 所示的与或阵列，再到功能强大的 MUX[1]。这是由于最简单的二选一 MUX 的功能要比二输入与非门或者是与门、或门所能实现的逻辑函数个数要多。具有 n 个选择端的 MUX 单元的输入个数为：$n+2^n$。因此 MUX 的输入端数随着选择端数 n 的增加而指数上升。例如当 n 为 3 时，输入端数为 11，如图 2-9(a)所示；当 n 为 4 时，输入端数为 20。从输入端个数方面考虑，作为可编程逻辑单元的 MUX 选择端个数一般都小于 3；但是作为可配置互连的 MUX 单元的选择端数一般都要大于 3。这是由于一般可编程器件中通道内的互连线段数目一般都超过 8。

图 2-15 是文献[13]中提出来的基于三个二选一 MUX 的可编程组合逻辑单元，其中第二级的 MUX 的选择端是由一个或门进行控制。商用可编程逻辑器件中也有采用类似结构的产品，这种单元是八个输入一个输出的结构。如果把选择端 S_a 和 S_b 合并成一个选择端，那么这三个二选一 MUX 就构成了一个四选一 MUX。没有合并 S_a 和 S_b 的原因是为了充分利用底层的硬件资源，提高电路实现的灵活

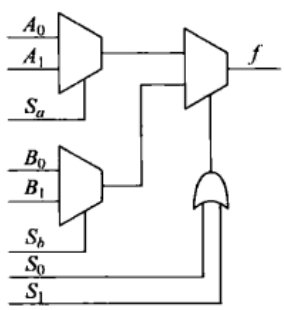


图 2-15　基于二选一 MUX 的可编程组合逻辑单元

性，从而实现四选一 MUX 所无法实现的函数。该论文指出这种结构能够实现所有的二输入、大部分的三输入和较多的四输入函数等。对于没有用到的可编程单元输入端口，论文指出只需要让这些输入端置为常数值即可。由此可见，图 2-15 所示具有八个输入端的可编程逻辑单元在实现少变量函数时，例如变量数小于 4 时，它的输入端口资源是非常浪费的。这就意味着部分输入端是作为常数信号使用的。这些常数信号会占据部分的局部互连资源，从而造成了互连资源的浪费。

利用 MUX 作为可编程单元的缺点是多选择端的 MUX 输入端个数太多，在实现小函数时就造成晶体管资源的浪费。如果基于二选一的 MUX 来构造可编程单元，那么只能实现小规模函数，实现较大函数时就会用到多个可编程资源，从而给互连资源带来很大压力。例如图 2-15 所示的单元就不能实现所有的四输入函数。因此我们需要在 MUX 的基础上寻找一种输入个数不是很多，但是容易实现规模稍大的逻辑函数，例如起码能很方便地实现所有的四输入函数。

突破 MUX 单元输入端数多，晶体管利用率不高的一种途径是让它的部分输入端不是作为用户数据，而是作为配置数据使用。这样就可以减少可编程单元的输入数，从而降低了互连的压力，同时又可以通过配置数据保证它的灵活性，能够实现多种函数功能。比较图 2-9 中的八选一 MUX 和三输入 LUT 的输入输出端口，就可发现如果保留 MUX 中的三个选择端，s_2、s_1、s_0 作为电路的输入端供用户使用，而把其余的八个输入端 x_7，x_6，…，，x_0 作为配置数据，那么八选一 MUX 就退化成为了三输入 LUT。这样输入端数从 11 个降到 3 个，大大减小了可编程单元外部的互连资源压力。下面再来分析一下 n 输入 LUT 的灵活性，即它所能实现的函数个数。

一个 n 输入单输出的 LUT 共有 2^n 个配置点，每个配置点都可以根据电路功能需求设置为 0 或者 1，于是 2^n 个配置点共有 2^{2^n} 种可能的配置值，其中每一种配置值都对应一个函数的真值表，因此一个 n 输入的 LUT 共能实现 2^{2^n} 个逻辑函数。对于二进制函数来说，n 变量的组合逻辑函数一共有 2^{2^n} 个。这就是说，一个 n 输入的 LUT 能够实现任意的 n 输入函数——只要把函数的真值表作为配置值就可。由此可见，LUT 的灵活性是非常强大的。同 MUX 相比，它的输入端口数目明显减少，从而降低了对可编程单元外部互连资源的消耗。另外也可以从存储器的角度来理解 LUT 的灵活性。LUT 的 n 个输入端可以看作是输入地址，那么一共有 2^n 个地址值。对于每个地址值，都可以存数据为 0 或者 1。这一位的数据就可以从 LUT 的输出端得到。这也是 LUT 被称为“查找表”的原因：给定一个地址，都能得到用户所存的一位数据。LUT 本身不对 2^n 个地址选择有限制，它同样不对每个地址上要存的 0、1 数据有限制，因此共有 2^{2^n} 种地址数据配置方式，每种配置方式对应一个逻辑函数。以下以 LUT3 为例说明如何确定任意 3 变量函数的配置值。

图 2-16 所示为三输入 LUT 的符号，任意三变量函数的真值表和晶体管级电路。三变量函数的真值表共有 $2^3=8$ 个项，分别对应于 000，001，…，111 共 8 种输入值，相应的函数值分别为 F0，F1，…，F7。在它的晶体管实现电路中，除了三个反相器外，共有 14 个 NMOS 传输管。三个输入变量 a、b、c 和它们的反相器输出分别接到传输管的栅极，控制它们的“截止”或者“导通”状态。当传输管的栅极为高电平时，管子就导通；反之，如果栅极为低电平，管子就截止。另外 LUT3 有 $2^3=8$ 个配置点，分别连到 8 个 SRAM 存储单元，用于存放配置信息。由于三个输入 a、b、c 只能取 000，001，…，111 中的一个值，因此它们分别控制八条

从 SRAM 配置端到输出端 f 的一条通路。例如，当输入 a、b、c 为 111 时，传输管 1、传输管 9 和传输管 13 就同时导通，这样只有最上面的那个 SRAM 配置值通过 f 输出，形成一条输入到输出的通路。其余的七条通路都是截止的。因此最上面的那个 SRAM 就对应于函数真值表中的 F7，即输入为 111 时的函数值。依此类推，SRAM 配置值从上到下分别为 F7、F3、F5、F1、F6、F2、F4、F0。于是任意三输入函数只要把真值表放到 SRAM 的配置点中就能利用 LUT3 的结构来实现。在实际应用中，由于传输管的输出电压存在阈值损失，因此 f 的输出一般经过一个反相器进行电平恢复。

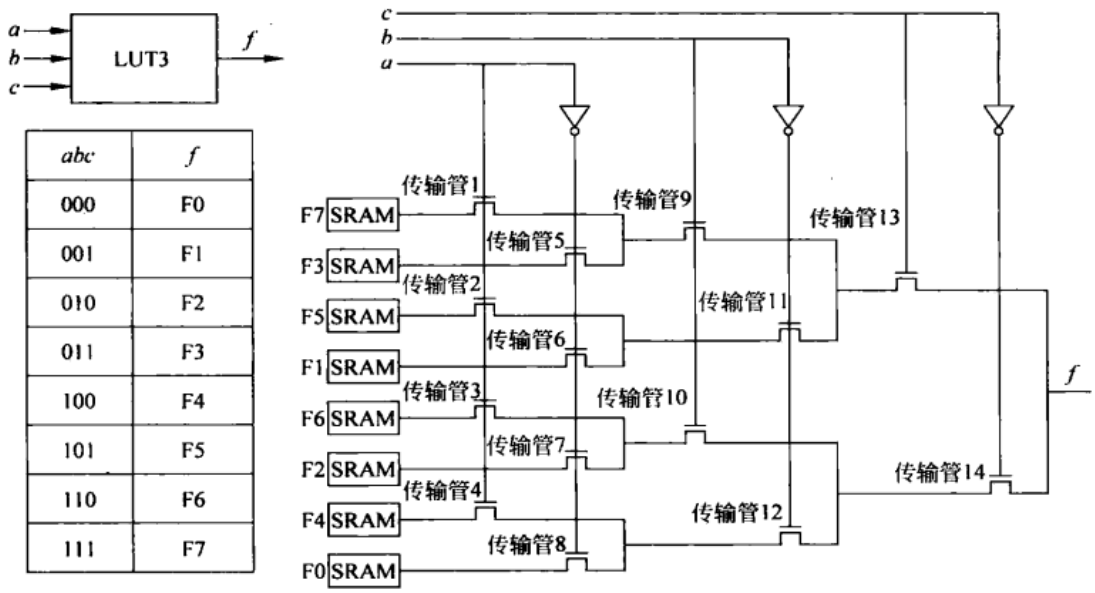


图 2-16　三输入 LUT 的符号、函数真值表、晶体管级电路和 SRAM 配置值

比较图 2-8 中二选一 MUX 和图 2-16 中 LUT3 的结构，可以发现两者的传输管实现电路结构完全一样。我们可以基于二选一 MUX 的单元构建 LUT3 晶体管级电路，只不过把 MUX 中的输入端 x_0、x_1 用 SRAM 配置单元值来代替输入值，并且三个连到输入端的反相器各自合并在一起即可。

虽然 n 输入的 LUT 结构能够实现任意的 n 变量单输出函数，但是对于 n 取何值，才能保证电路的面积和性能最优呢？一方面，如果 LUT 太小，那么实现变量数超过它的输入数时就需要占用可编程外部的互连资源，并且电路延时较大；另一方面，如果 LUT 的输入数太多，那么对于实现变量数小于它的输入数的函数时，LUT 的晶体管资源就会浪费，并且所经过的晶体管的延时级数还不能减少，这同样会影响硬件资源利用率和延时性能。对于这个基本的可编程单元结构问题，文献[14]已经提供了合理的答案。根据当时的工艺水平和基准测试电路规模，该论文指出输入端个数为 4 的 LUT 结构能够达到最优的面积利用率和时序性能。因此主流商用 FPGA 器件的可编程逻辑单元都是把 LUT4 作为基本单元，尽管有时 LUT4 可以拆分为两个 LUT3 使用，或者两个 LUT4 合并成为一个 LUT5。当半导体工艺水平达到 65nm 节点，晶体管的集成度进一步提高，这样可以把更多的电路功能放到一个 LUT 中去，从而把本来要占用可编程单元的外部互连资源都释放出来。因此主流 FPGA 器件在 65nm 以后就采用了 LUT6 作为基本单元。

由于实际应用电路大部分是时序电路，纯组合电路的比例很少。因此在电路中一般都需要触发器或者寄存器。虽然从功能实现上来说，利用 LUT 可以实现触发器，但是文献[14]的研究表明没有触发器作为基本单元的可编程逻辑器件所需要的互连资源将是带触发器可编程单元器件的两倍以上。图 2-17 就是文献[14]所提出的基于 LUT 的可编程逻辑单元基本结构模型。其中上面一个二选一 MUX 用于选择 LUT 的组合输出或 DFF 的时序输出。最后的三态门用于电路测试。通过下面一个 MUX 可以让三态门断开，或者由外面的“使能输入”进行动态控制。两个 MUX 之间的“State”就表示配置值的状态。现在测试电路一般都用扫描链进行解决，而不是用三态门。我们可以说现有主流商用 FPGA 器件中的可编程逻辑单元本质上都没有摆脱这种结构。

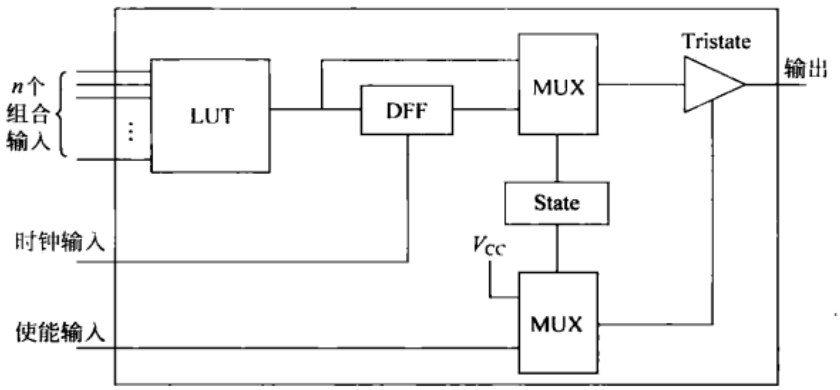


图 2-17 基于 LUT 的基本可编程逻辑单元

2.3.2 可编程互连结构

可编程互连就是指可编程逻辑单元、可编程 IO 单元之间的连线资源，如图 2-2 中的连接盒、开关盒或者其他更为复杂的结构。可编程逻辑单元或者 IO 单元的输入输出端口先经过连接盒连到水平或者竖直通道的互连线上，然后通过一个或多个开关盒进行互连线的切换、拐弯，最后再经过连接盒到达另外一些可编程逻辑单元或 IO 单元的输出输入端。这些连接方式同样都是由配置信息控制的。通过合适的配置，多个可编程逻辑单元通过互连资源能够实现容量较大的数字电路。

可编程互连资源结构和可编程逻辑资源结构是紧密相关的。文献[14]在讨论可编程逻辑单元结构是否直接包含 DFF，而不是通过 LUT 来实现 DFF 的功能时，指出不直接包含 DFF 单元时，可编程逻辑单元的数目大约要增加 1.4～2.3 倍。但是如果用 LUT3 或者 LUT4 来实现 DFF 时，不直接包含 DFF 的可编程逻辑单元面积比包含 DFF 的逻辑单元面积大约要小 2.1～2.5 倍，因此两者的逻辑单元面积基本持平。但是对于实现同一个电路来说，不包含 DFF 的可编程逻辑单元所需要的个数要多一倍左右，因此所需要的互连资源就比直接包含 DFF 的可编程单元多一倍，尽管它们的可编程逻辑单元面积基本相同。因此综合考虑可编程逻辑资源和可编程互连资源的需求，直接包含 DFF 的可编程逻辑单元比不包含 DFF 的可编程单元对应的 FPGA 综合性能要好[14]。

根据 2.3.1 节讨论可编程逻辑单元时对 MUX 结构的分析结果，我们知道 MUX 单元非常适合于实现互连资源的连接选择。对于具有 n 个选择端的 MUX 单元来说，它共有

$n+2^n$ 个输入端，其中 n 个用户选择端可以控制 2^n 个输入信号和输出之间的连接关系。如果把 2^n 个输入信号作为配置点输入，n 个选择端作为用户输入，那么 MUX 就退化成了 LUT 单元。如果把 n 个选择端作为配置点输入，2^n 个输入信号用于传输用户信号，那么这个 MUX 就用于控制本节所讨论的可编程互连资源的连接关系。如图 2-18 所示，一个八选一的 MUX 通过三个配置点的控制，就可以在八条互连线之间进行不同的连接选择。图中每个 SRAM 配置点有两个互补输出，分别控制各传输管的状态。比较图 2-18 中基于 MUX 的互连控制方式和图 2-16 中 LUT 的晶体管级电路，我们可以发现两者的主要差别在于配置点和用户输入的位置。图 2-16 中的反相器输出就用 SRAM 单元中互补的两个输出实现。SRAM 单元的两个互补输出见图 2-12。因此从晶体管的角度来看，图 2-18 左边的一个配置点控制单个传输管的方式既可以用于可编程逻辑单元 LUT 的电路实现，也可以用于互连资源的连接控制。晶体管本身并不能区分它是被用于逻辑计算，还是互连选择。基于这样的理解，可编程逻辑单元结构和可编程互连单元结构之间的界限就比较模糊了。文献[7]所提出的 MUX 能够在可编程逻辑和可编程互连之间进行切换的可编程单元结构也进一步说明了这种模糊性可能给 FPGA 结构设计带来新的思路。

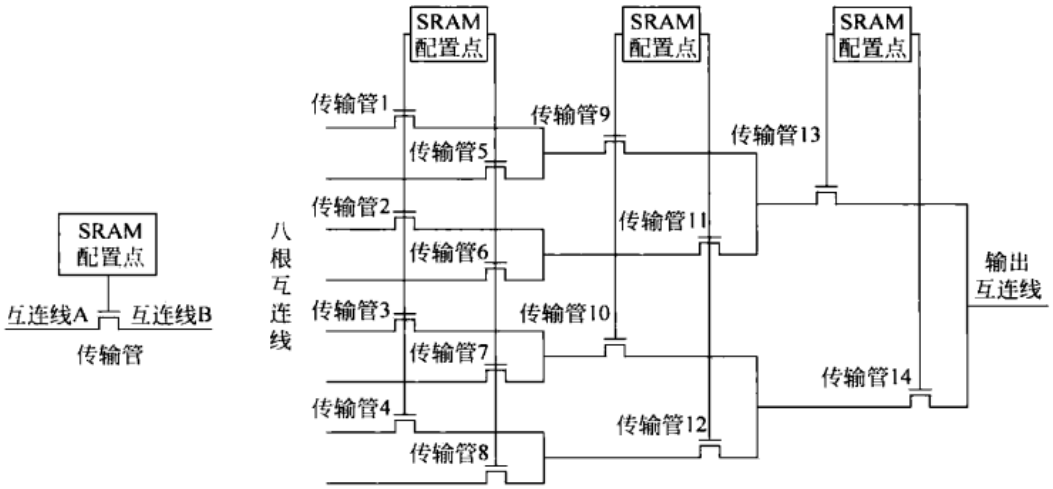


图 2-18　基于单管和 MUX 的互连控制方式

基于以上传输管和 MUX 的互连可编程性，我们就可以设计不同结构的可编程互连资源。单个传输管就可以控制两条互连线之间的连接或断开方式，其优点是双向导通，成本很低；主要缺点是存在阈值损失，驱动能力不强。多个级联的传输管通路一般都要接反相器等电平恢复或整形电路。多条连线之间的连接控制就用 MUX 来实现。虽然它只能实现从输入到输出之间的单向连接，但是速度快，配置码点个数少，比传统上基于三态门的总线连接方式性能好。

利用以上 MUX 或单个传输管就可以实现不同的互连方式，然后在此基础上进行结构优化，从而得到不同的互连拓扑结构。传统 FPGA 的互连结构是基于连接盒或者开关盒，如图 2-19 所示。图中画出了 2×2 逻辑单元阵列的部分连接关系，其中假设可编程逻辑单元共有四个输入输出端口，均匀分布在四边，每个“×”表示通过传输管控制的连接关系。连接盒用于控制可编程单元输入输出端和互连线之间的连接关系。由于可编程单元的输入输

出数目一般有几十个，并且互连线的数目一般也有几十条，因此它们就不适合于用单个传输管控制两两之间的连接关系，否则配置点的数目会很多。此时如果用 MUX 就可以明显节约晶体管的数目。例如我们可以假设有 n 个连接关系需要控制，对于传输管的控制方式，那么共需要 n 个配置点。如果采用 MUX 的控制方式，那么只需要 $\log_2 n$ 个配置点，因此连接盒的连接关系常用 MUX 来控制。连通度是连接盒的主要结构参数它表示一个可编程逻辑单元端口能够和通道中互连资源相连的互连线条数或者比率。例如图 2-19 中一个逻辑单元端口和四条竖直互连线中的三条有可配置的连接关系，那么竖直连通度就是 3/4＝75%。它和水平互连线中的两条有可配置的连接关系，那么水平连通度就是 2/4＝50%。连接盒的竖直和水平连通度可以不同，也可以相同，并且可以随着在阵列规模中的不同通道位置而改变。根据文献[8]的研究结果，一般水平和竖直通道的连通度相同时该 FPGA 结构的综合性能较好。

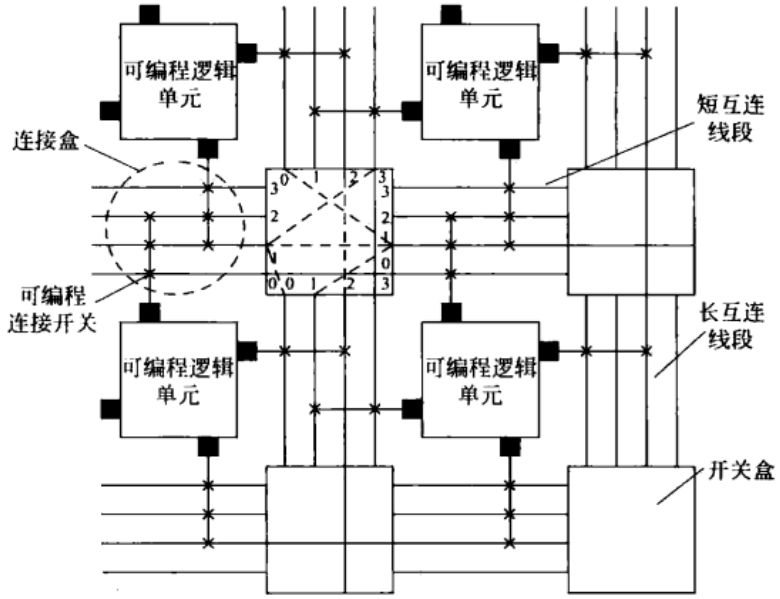


图 2-19 基于连接盒和开关盒的 FPGA 互连结构示意图[8]

开关盒主要用于定义竖直互连线和水平互连线之间的可配置连接关系，在图 2-19 中这些可配置的连接关系用开关盒内部的虚线来表示。它的连接关系主要由拓扑结构来决定。对于四边形的开关盒来说，一个边上的互连线可以和其余三边的一条互连线相连，即它可以往两边拐弯，同时也可以直连。因此传统开关盒的连通度为 3，即一条互连线可以和三个方向的互连线相连，但是它们的连接拓扑结构可以不同。

图 2-20 所示为三种典型的开关盒拓扑结构，分别为 Disjoint（分离型开关盒，也叫 Subset 型开关盒）[15]、Universal[16]和 Wilton[17]。它们的连通度都是 3，即一边上的互连线均能连到另外三边的互连线，其中包括直连到正对边的连接方式。它们的主要差别是连线的“域”不同，从而导致电路性能的差异。如图 2-20 所示，开关盒四边的每条互连线都有一个标号：“0”、“1”、“2”、“3”，每一个标号对应于一个“域”。在 Disjoint 结构中，以左边的“3”号互连线为例，它只能连接到其他三边中标号为“3”的互连线。也就是说，互连线不能通过开关盒实现跨域连接。这就会影响布线的灵活度以及布线后电路的时序性能。在

Universal 结构中，它已经能够实现跨域连接，例如左边“3”号互连线可以连到上边的“0”号互连线。但是其他两边上的互连线还是同“域”连接，即它不能连到右边和下边的标号不是“3”的互连线。Wilton 开关盒结构则进一步实现了跨域的连接。同样以左边的“3”号互连线为例，它能够直连到右边的“3”号互连线，另外它还可以连到上边的“1”号互连线和下边的“2”号互连线。因此，从跨域连通的灵活度来说，Wilton 结构相对比较灵活。

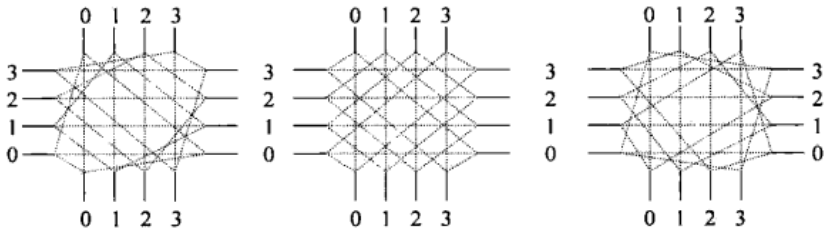


图 2-20　三种不同拓扑结构的开关盒：Disjoint、Universal 和 Wilton

对于给定的通道宽度 W，表 2-2 画出了 Wilton 结构所定义的开关盒连接关系。具体来说，这种连接关系表示为从表格中第一列所列标号为 x 的互连线能够连接到第一行各边所列的互连线的标号关系。可以注意到由于连通度为 3，因此表格中的对角线方向都没有连接关系，即一条互连线不能连接到同一边上的其他互连线。从布线性能的角度来看，这种从开关盒的一边进来，又从同一个开关盒的同一边逆向出来的连接关系是比较少见的。另外所有的直连边的互连线标号均相同。例如在表格中每一行都有一个标号为 x 的值，它们的连接关系从第一行到第四行依次为：左边→右边，右边→左边，上边→下边，下边→上边。最后我们可以发现 Wilton 结构的连接关系是对称的，即如果某一边的一条标号为 x 的互连线能够连到另一边编号为 y 的互连线，$0 \leqslant x, y < W$，那么一定存在反方向的连接关系。例如表格中“左边→上边”和“上边→左边”的连接关系均为 $(W-x)\%W$，其中“%”表示“取模”运算，它和 C 语言中的定义是一样的。或者更具体地说，假设左边标号为 x 的互连线能够连接到上边标号为 $y=(W-x)\%W$ 的互连线，那么上边标号为 y 的互连线一定能够连接到左边标号为 x 的互连线。用公式表示如下：

$$
\begin{aligned}
& y = (W-x)\%W \\
\Rightarrow \\
& (W-y)\%W \\
& = (W-(W-x)\%W)\%W \\
& = (W-W+x)\%W \\
& = x
\end{aligned}
\tag{2-4}
$$

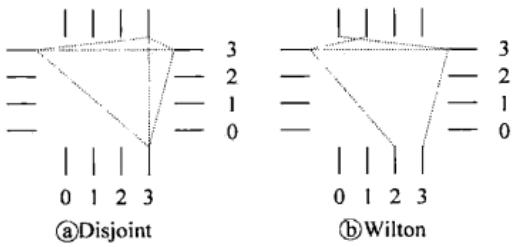
表 2-2　Wilton 开关盒的连接方式

互连线(↗)	左边	右边	上边	下边
左边	—	x	$(W-x)\%W$	$(W+x-1)\%W$
右边	x	—	$(x+1)\%W$	$(2W-x-2)\%W$
上边	$(W-x)\%W$	$(W+x-1)\%W$	—	x
下边	$(x+1)\%W$	$(2W-x-2)\%W$	x	—

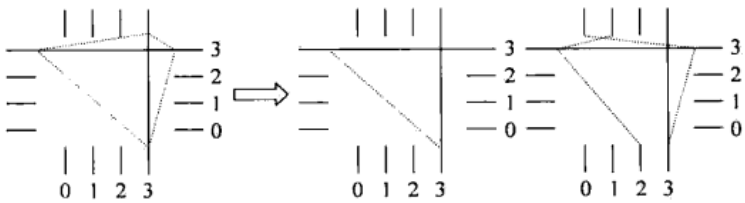
同理，对于表格中“右边→上边”和“上边→右边”的互连线标号关系分别为$(x+1)\%W$和$(W+x-1)\%W$。更具体地说，假设右边标号为x的互连线能够连接到上边标号为$y=(x+1)\%W$的互连线，那么上边标号为y的互连线一定能够连接到右边标号为x的互连线。用公式表示如下：

$$\begin{aligned} & y=(x+1)\%W \\ \Rightarrow & \\ & (W+y-1)\%W \\ & =(W+(x+1)\%W-1)\%W \\ & =(W+x+1-1)\%W \\ & =x \end{aligned} \tag{2-5}$$

以下我们说明从电路面积和时序性能考虑，Wilton结构的开关盒其实不一定比其他两种都要好。对于互连线长度都是1的单倍线FPGA结构来说，同样以图2-20中左边和右边的两条“3”号互连线为例，它们在每个开关盒中都有3种可配置连接，其中有一个连接是重合的，因此共需要5个传输管进行控制，不管是分离型或者Wilton开关盒。为了简化起见，图2-21(a)单独画出了“3”号互连线在分离型和Wilton两种开关盒内部的连接关系。可以看出在图2-21(a)中，两种开关盒内部各有5条可配置的连接，其中从左边“3”号线连到右边“3”号线的配置关系和从右边“3”号线连接到左边“3”号线的配置连接是重合的，因此只需要5个传输管来配置。但是对于互连线比单倍线要长的FPGA结构来说，分离型和Wilton开关盒所需要的传输管个数则不同。因为对于非单倍线长度的互连线来说，它们往往会直通过开关盒，而不会在每个开关盒中都需要3个配置连接。如图2-21(b)所示，对于这两条标号为“3”的互连线来说，分离型开关盒水平方向的“3”号线会直通过开关盒，竖直方向的开关盒也会直通过开关盒，这是因为它们的互连长度大于“1”，能够直接连接距离跨度较大的可编程逻辑单元。这样对于分离型开关盒来说，本来需要5个传输管配置的连接关系，现在只需要1个了。详细地说，左边“3”号线到上边“3”号线的配置连接、右边“3”号线到上边“3”号



(a) 单倍线时分离型开关盒和Wilton开关盒的连接关系



(b) 非单倍线时分离型开关盒和Wilton开关盒的连接关系

图2-21 分离型开关盒和Wilton开关盒的性能比较

线的配置连接、左边“3”号线到下边“3”号线的配置连接、右边“3”号线到下边“3”号线的配置连接由于直通互连线的缘故只需要一个传输管就够了。这就是说，原来分开的4个配置连接现在都重合在一起了，这样原来的4个传输管现在只需要1个就够了。但是对于Wilton结构来说，由于它们是跨域连接，因此直连线并不能节约传输管的数目，还是需要4个传输管才能实现Wilton结构的互连方式，如图2-21(b)所示。因此从晶体管的个数来考虑，包括配置传输管和SRAM配置单元所需要的晶体管数目，分离型开关盒能比Wilton开关盒节约很多的晶体管。具体数目可以通过本章习题8进行比较。虽然Wilton结构比分离型结构能够在布线方面有更多的自由度，但是它所需要的晶体管数目要明显多于分离型开关盒，它所增加的面积成本已经足够抵消它的布线灵活性所带来的好处。另外由于一条互连线所连接的晶体管数目增加，从而加大了互连线的负载，于是基于Wilton结构的电路时序性能也会下降。因此综合考虑面积和电路性能，分离型开关在互连线段长度超过单倍线时比Wilton结构要优越，文献[8]的研究结果证实了这种理解的正确性。

基于连接盒、开关盒的FPGA互连结构是早期商用FPGA器件所采用的结构，例如Xilinx公司X4000系列。这种结构的主要缺点是互连延时较大。一条简单的线网从可编程逻辑单元的LUT或者DFF的输出端必须先经过连接盒才能连到互连资源，然后至少要经过开关盒中的一条直通互连线，再经过另一个连接盒才能连到另一个可编程逻辑单元的输入端。如果要实现长距离的连接，那么所经过开关盒的可编程互连开关就很多。为了减少这种结构的互连延时，对于短距离连接，可以增加快速直连可编程互连，即可编程逻辑单元的输入输出只要经过一个传输管就可以连接到与之相邻的可编程逻辑单元的输入端。对于长距离连接，一种方法是在连接盒和开关盒的基础上引入层次化的互连结构。还有一种方法是不再区分连接盒和开关盒，把连接盒和开关盒统一为一种更为灵活的互连结构，从而摆脱了区分连接盒和开关盒的束缚[9]。以下分别介绍这两种互连结构。

从互连资源的结构需求来看，由于用户电路的连接关系结构一般表现出一定的局部性，即整个电路可以划分为多个子模块，各子模块之间的连接关系比子模块内部的连接关系要少，基于这种直觉，可编程互连资源结构就自然地采用层次化结构。这种结构类似于日常生活中的城市内和城市间的公路结构。城市之间高速公路是不会有红绿灯进行交通控制的，而城市内的低速公路则有很多的交叉路口，便于变换方向。图2-22是文献[18]提出的一种快速互连结构，图中的正方形小方框表示可编程逻辑单元，互连资源一共分为两层，底层的可编程逻辑单元经过连接盒和开关盒实现层内局部连接。对于远距离连接，它必须经过上一层的可编程互连资源实现，例如图中的粗线所示的通路。这种长距离互连线中间可能存在可编程触发器，便于信号的流水线处理。这种结构的优越性在于高效的远距离连接；缺点是有时即使两个可编程单元的物理距离相隔很近，但是它们在布局时由于逻辑容量的限制可能被分配到了不同的底层单元，例如图中左边标记为“1”和“2”的两个可编程逻辑单元，虽然它们靠得很近，但是还必须要经过上层的互连和底层的连接盒和开关盒才能连通，如图中的粗线所示。早期的商用可编程逻辑器件经常采用这种层次化互连结构，例如Altera公司的APEX系列器件等[11]。

需要注意的是上述层次化互连结构和一般所讲的局部互连和全局互连的层次化是两个不同的概念。所谓局部互连一般是指包含多个基本可编程逻辑单元的逻辑簇内部的互连资源；而全局互连是指能够连接到芯片内部任意可编程逻辑单元的时钟树。时钟树除了对速

度有很高的要求，更重要的是同步性要求，保证所连接到的触发器能够同步触发，从而能够提高时钟的频率，并保证电路的稳定工作。

上述层次化互连结构的缺点是仍保留了连接盒和开关盒的结构，使得相邻可编程逻辑单元之间最简单的连接方式也必须经过两个连接盒和一个开关盒。如果我们能够把连接盒和开关盒都归结到一个通用互连单元，从而不再有独立的连接盒和开关盒，那么它就会有更高的灵活性。例如图 2-23 把可编程逻辑单元的输入输出端口和互连线都直接连到可编程通用互连单元，那么逻辑单元的输入输出就可以直接在这些单元内部[9]。上述层次化互连结构中为短距离连接所添加的快速直连结构可以看成是这种结构的一个特例。通过这种通用互连单元结构，可编程逻辑单元之间、可编程逻辑单元和换向的互连线之间都可以直连，而不像之前的结构，一定要先经过连接盒才能在开关盒中换向。与上述层次化互连结构相比，这种互连结构不再有分层的互连资源，所有的通用互连单元都相同。当然连接到通用互连单元的互连线种类、数量和长度是多种多样的。虽然这种结构要比单个连接盒和开关盒复杂，端口数增加，并且所需要的晶体管多，但是它所能提供的灵活性和简洁性更具有吸引力。另外由于半导体工艺技术的进步，晶体管资源已经变得非常便宜。芯片的速度和功耗为优先考虑的因素。如果通过增加晶体管能够提高可编程逻辑器件的灵活性和性能，那么这是一个非常自然的实现途径。例如商业的 Xilinx 公司 Virtex-Ⅱ之后的互连结构都是采用了 GRM(general routing matrix)结构，而不再区分连接盒和开关盒。

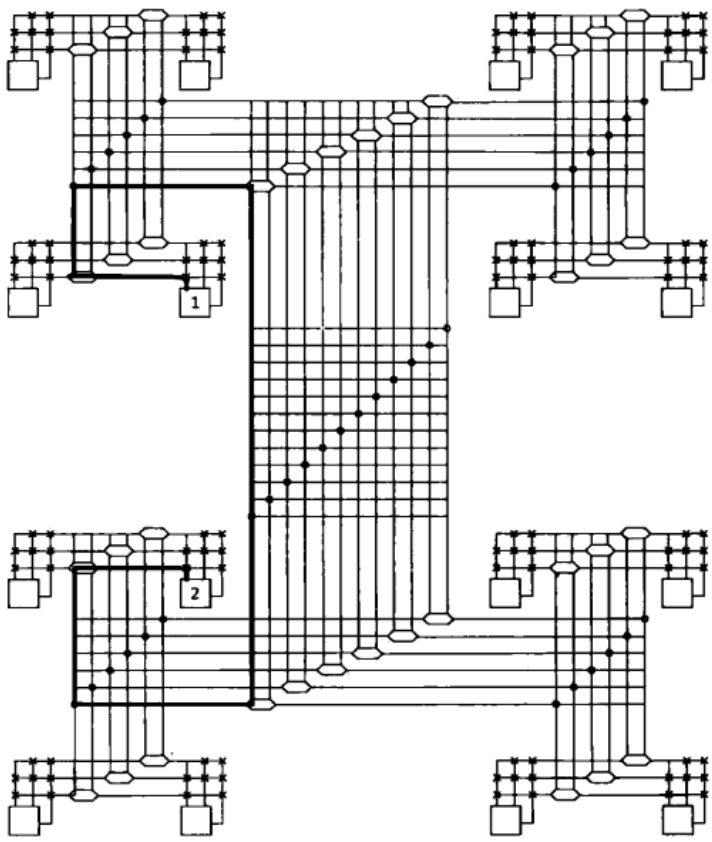


图 2-22 基于连接盒和开关盒的层次化互连结构[18]

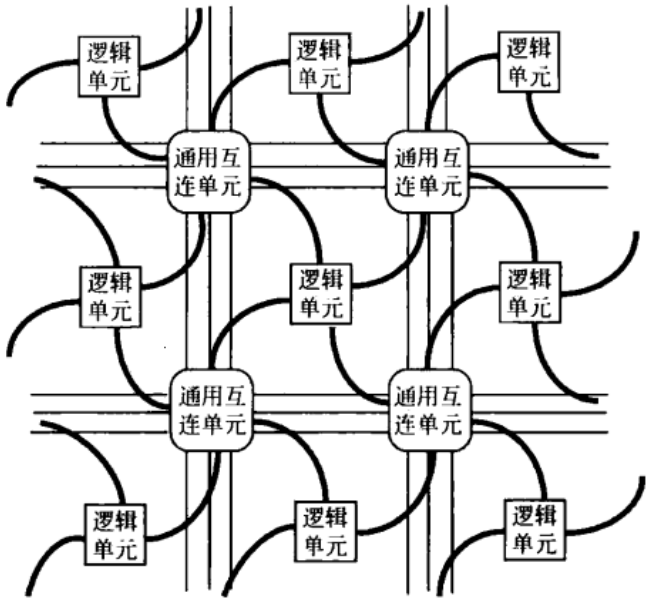


图 2-23　统一连接盒和开关盒互连结构示意图

随着半导体工艺技术的继续发展，芯片集成度和面积越来越大。可编程互连资源所占的面积超过整个芯片面积的 70%，互连延时一般占关键路径延时的 70%以上，对可编程互连资源的结构研究就有更大的需求。对于传输距离超过 1mm 的互连线来说，虽然可以通过增加缓冲器甚至是触发器进行流水处理，从而尽可能地减少延时，但是还应该寻求更为有效的结构。一种可行的方法是借鉴网络技术，就是把片外高速通信技术应用到芯片内部，即 NoC(片上网络)技术。这种技术不只是通过物理级的金属层来传输信号，更重要的是它引入了数据包的传输方式，从而实现远距离的快速传输。例如图 2-23 所示的互连方式就非常类似于网路拓扑中的 Mesh 结构[19]。我们相信 NoC 技术能够给可编程互连结构研究提供新的方向。

2.3.3　可编程 IO 单元

作为可编程逻辑器件和片外电路的接口，可编程 IO 是可编程逻辑器件的基本组成单元。出于灵活性的考虑，可编程 IO 单元必须实现基本的功能：带寄存器和不带寄存器的输入，带寄存器和不带寄存器的输出，双向 IO，并且要支持多种电平标准等。

图 2-24 是具备简单输入输出的 IO 功能示意图。首先它具有一个统一的对外端口，即图中右边标记为“端口”的部分。不管采用何种配置模式，这个端口都是共享的。左上方的“输出使能”端提供给用户控制端口的输入输出方向。如果该信号为 1，那么这个端口作为输出使用；反之，如果为 0，那么这个端口作为输入使用。这个控制信号本身又是可配置的：带寄存器和不带寄存器的信号控制方式。该图中共有三个配置码点，分别用于端口方向控制信号、输入和输出三个信号的组合和时序模式控制，因此这三个信号的寄存器模式控制就需要三个寄存器。如果要让用户还能把端口配置为双向模式，那么另外还需要添加配置点和输出使能信号端口。图中用粗线画出了各种 IO 控制方式和信号流向，包括“组合输入”、“组合输出”、“带寄存器输入”、“带寄存器输出”，并且还能实现电路工作时能改变方向的双

向 IO 功能。另外出于芯片测试的需要，一般可编程 IO 单元还带有边界扫描的功能，用于支持 JTAG 的自动测试技术。关于 JTAG 的介绍参见 3.6 节。

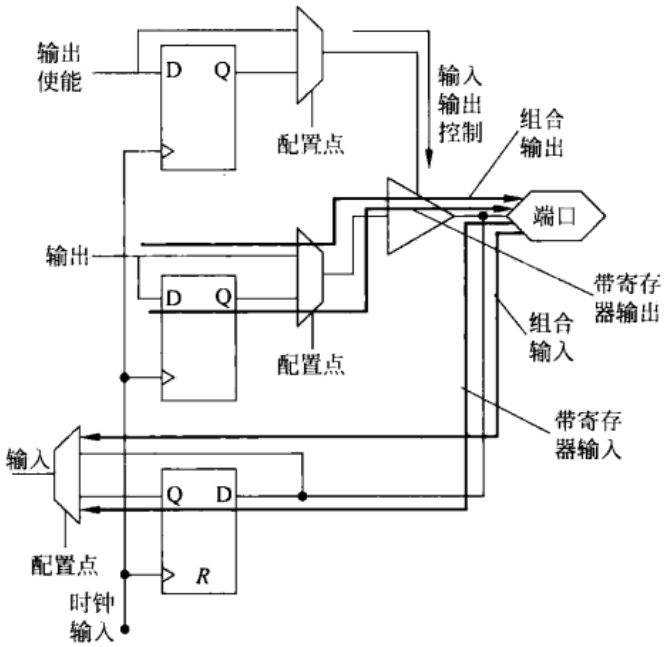


图 2-24 可编程 IO 单元的基本功能示意图

FPGA 作为通用的数字电路实现器件，其接口需要支持多种电平标准。如 1.6 节所介绍的，这种性能使得可编程逻辑器件能够作为“胶合逻辑”的基础。表 2-3 列出了常用的数字电路输入输出电平标准。TTL(晶体管晶体管逻辑)和 CMOS 为经典的数字电路电平标准。由表中数据可见 TTL 和 CMOS 的高低电平差值较大，因此电路需要较长的时间进行翻转。另外它们的电源电压偏高，从而导致较高的功耗，因此就出现了低电压的电平标准：LVTTL(低电压 TTL)和 LVCMOS，它们分别有 2.5V 和 3.3V 的两种低电压标准。电源电压的下降，为低功耗设计带来很大益处。为了进一步提高信号的抗干扰能力，能够有效地抵消在信号传输过程中的共模信号，就引入了 LVDS(低电压差分信号)差分电平。它的电源电压为 2.5V，两个相位互补的信号摆幅降到 350mV，因此可以传输高速信号。表格中的 HSTL(high-speed transceiver logic，高速收发逻辑)和 SSTL(stub series terminated logic，短线串接逻辑)分别是需要比较稳定的参考电平输入的高速差分电平标准，它们的参考电压值一般是电源电压的一半。

表 2-3 常用可编程 IO 单元支持的电平标准 (电压单位：V)

名 称	电源电压	输出高电平下限	输出低电平上限	输入高电平下限	输入低电平上限
TTL	5.0	2.4	0.5	2.0	0.8
CMOS	5.0	4.5	0.5	3.5	1.5
LVTTL_3.3	3.3	2.4	0.4	2.0	0.8
LVTTL_2.5	2.5	2.0	0.2	1.7	0.7

续表

名　称	电源电压	输出高电平下限	输出低电平上限	输入高电平下限	输入低电平上限
LVCMOS_3.3	3.3	3.2	0.1	2.0	0.7
LVCMOS_2.5	2.5	2.0	0.1	1.7	0.7
LVDS	2.5	±350mV的差分电平			
HSTL_1.8	相对参考电压0.9V的差分信号				
HSTL_1.5	相对参考电压0.75V的差分信号				
HSTL_1.2	相对参考电压0.6V的差分信号				
SSTL_2.5	相对参考电压1.25V的差分信号				
SSTL_1.8	相对参考电压0.9V的差分信号				

可编程IO单元除了支持功能的灵活性和电平标准的多样性外，电路性能的不断提高对它提出新的要求。例如为了达到IO输出和片外输入的时序匹配要求，可编程IO单元配置输出IO的驱动能力，延时补偿控制和可编程的上拉电阻等。总体上来说，可编程IO单元要比专用芯片的输入输出单元复杂很多，在灵活性、性能、可靠性和面积方面都有更高的要求。

2.4　系统级FPGA结构特点

2.3节主要介绍了基于细粒度可编程FPGA结构，包括基于LUT的可编程逻辑结构和基于互连线的可编程互连结构以及可编程IO单元。随着半导体工艺技术的不断进步，易扩展的FPGA阵列化结构使得它的容量不断增大，并行处理能力明显增强。另一方面，半导体工艺技术的进步使得专用芯片设计成本大幅上升。国际著名的评估机构BDTI (Berkeley Design Technology Inc.)从21世纪初的评估报告中明确指出系统级FPGA的并行处理能力远远高于传统的DSP处理器[20]。在数字信号处理系统中，FPGA器件经常和CPU、DSP等处理器配合，作为加速部件使用。由于自主设计的FPGA技术门槛很高，在FPGA器件内嵌入一些粗粒度的DSP核、微处理器核、存储器核等单元远远要比在通用处理器、DSP芯片、ASSP专用芯片内部嵌入FPGA核容易很多，因此FPGA逐渐从传统的胶合逻辑应用逐渐扩展到计算密集型的系统级应用领域。

系统级FPGA就是在传统的细粒度FPGA基础上嵌入了基于字节的粗粒度运算、存储和控制等单元，包含用于乘加的DSP模块，用于数据存储的存储器核和通用的微处理器核等。虽然FPGA内部的DSP模块功能和DSP处理器差距很大，但是一块FPGA芯片内部通常嵌入了上百甚至上千个DSP模块，因此这些DSP模块的并行计算能力远远超过了单核或者多核的DSP芯片。从存储的角度来看，虽然细粒度的LUT可以作为分布式存储器使用，但是这种存储器将消耗大量的LUT资源，并且存储性能和嵌入式存储器核相差很远，因此系统级FPGA内部一般都嵌入了规模不等、支持多种访问模式的可配置存储器单元。另外从使用便利性的角度来看，传统FPGA基本上还是面向数字电路的硬件设计人员。使用FPGA的设计难度要远远高于基于通用处理器和高级语言的编程难度。但是如果FPGA内部嵌入一个或多个通用微处理器核和配套的编译调试环境，那么FPGA所能提

供的多处理器核系统和定制化指令等优点吸引了更多的人员使用 FPGA 器件。以下各节分别介绍系统级 FPGA 内部的嵌入式粗粒度模块。

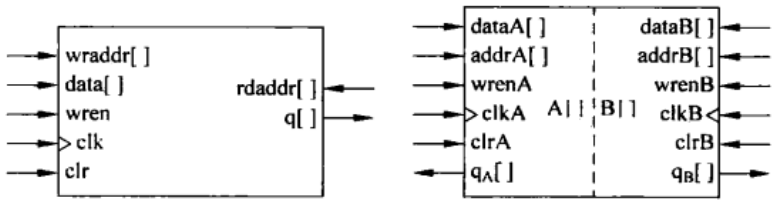
2.4.1 嵌入式存储器

如 1.2 节和 2.2.2 节所述，存储器技术一直是可编程性的基础。不管是通用型微处理器、数字信号处理器还是 FPGA 芯片，存储器一直是主要的组成部件。从图 1-15 存储器的层次结构可知，不同应用的可编程技术所需要的存储器单元结构、粒度和性能都不一样。例如 FPGA 内部的配置存储一般都是基于细粒度的 SRAM 单元，它并不需要同步时钟。CPU 内部的高速缓冲区(cache)一般也是采用 SRAM 单元阵列设计的。但是 FPGA 中用于实现数字逻辑电路的存储单元都是结构比 SRAM 单元更为复杂的寄存器或者触发器，它们都是需要有同步时钟的。虽然商用 FPGA 中也可以把查找表 LUT 作为分布式存储单元使用，但是分布式存储单元的性能和专用存储单元差很多。从功能上来说，存储单元也可以利用经典的与非门或者其他组合逻辑单元来实现，但是由于存储器是典型的可重复阵列化结构，这样对存储单元的面积和性能优化要求就很高。如果这种可重复的存储单元面积、性能、可靠性和成本都不是很好，那么一个包含成千上万个存储单元的存储器肯定会缺少竞争力，因此存储器单元结构一般都会做到极致，从而达到最高的优化效果。芯片制造商一般都会提供存储器编译工具(memory compiler)，用于帮助用户自动设计基于不同半导体工艺节点的存储器结构，并自动产生版图。对于 FPGA 来说，嵌入式存储器就不能局限于基于 LUT 的分布式实现方式，而是设计专门的可配置结构从而达到最大的优化效果和灵活性。

与专用芯片中的嵌入式存储器不同，FPGA 中的嵌入式存储器一般要满足通用性的需求。也就是说，FPGA 内部所提供的嵌入式存储器必须能够满足多种应用领域需求，例如它们能够提供不同数据宽度和深度以及多种工作模式。商用 FPGA 器件内部的嵌入式存储器一般支持 8 位、16 位、32 位、64 位等多种数据宽度和相应的地址深度，例如 32K、16K、4K、2K 等。它们的工作模式也有多种，例如单口、简单双口和真双口等。从性能的角度来看，FPGA 内部的嵌入式存储器必须采用专门的互连资源，从而保证它们的工作速度，而不会受其他用户逻辑的影响。

图 1-14 是单口嵌入式存储器的基本接口，它包含了基本的地址、数据输入和数据输出、写使能、时钟和复位信号。双口存储器的接口如图 2-25 所示。其中图 2-25(a) 是比较简单的双口格式，存储器可以在同一个时钟的同步下读写不同的地址单元。这种形式的存储器在一些数字信号处理的算法中是比较普遍的，在一个时钟内可以同时读写不同存储地址的数据，从而提高数据访问能力。功能更为强大的双口存储器(又称真双口存储器)如图 2-25(b)所示，和左边的简单双口存储器形式相比，真双口存储器具有两套独立的地址、输入输出数据、时钟、复位等信号。而这两套接口可以共享同样的内存空间。另外 A 端口和 B 端口允许有不同的数据宽度和深度。假设存储器容量为 nKB，其中“KB”表示一千字节的存储单位，即它具有存储 n 千字节的容量。如果把它配置为 16 位的双字节数据读取方式，那么地址空间就要减少一半，即为 $n/2$。如果配置为 32 位的数据读取方式，那么地址空间再减少一半，即为 $n/4$。例如 n 为 16 时，那么 16KB 的存储器通过外加一些控制逻辑，在 FPGA 器件内部可以配置为 8K 地址空间的 16 位数据模式，或者是 4K 地址空间的 32 位数据模式。这些多种配置的存储器读取模式是以外加简单控制逻辑为代价的，用于提高 FPGA 硬

件资源利用率和灵活性。商用 FPGA 器件一般包含阵列化的多种规模和工作模式的嵌入式存储器核，这些核之间还可以进一步根据用户需求合并为更大的存储单元，使得存储器的灵活性和利用率进一步提高。但是如果用户的电路中没有任何存储单元，那么这些嵌入式存储器资源就浪费了。



(a) 简单双口　　(b) 真双口

图 2-25　双口存储器的两种形式

2.4.2　嵌入式微处理器软硬核比较

FPGA 传统上作为一种通用的数字电路设计平台，它的普及度和便利性远比不上基于通用处理器及其编译调试工具的软件设计平台。一般来说要用好 FPGA，必须掌握硬件描述语言或者电路原理图等基本的数字电路设计知识。这种基于硬件并行化的设计方法还没有基于高级语言的串行化编程方法容易掌握。因此，如果能够在 FPGA 内部嵌入通用型微处理器核，不管是硬核还是软核，那么 FPGA 的使用便利性和应用领域就会得到很大的扩展。

对于 FPGA 来说，由于存在通用型 LUT 单元的数字电路实现方式，软硬核概念和专用集成电路有所不同。从 LUT 的通用性来说，嵌入式微处理器的功能可以通过 LUT 和触发器来实现。但是基于 LUT 实现的嵌入式微处理器性能较差，无法和直接嵌入到 FPGA 的微处理器硬核的性能相比。我们把基于 LUT 等通用逻辑资源实现的嵌入式微处理器称为软核；而在制造 FPGA 器件时直接把微处理器版图放入到 FPGA 内部的实现形式称为硬核。在商用 FPGA 中，有把 PowerPC 和 ARM 作为嵌入式硬核的器件；也有 MicroBlaze、Nios 和 ARM 的嵌入式微处理器软核。嵌入式微处理器的具体结构介绍见第 4 章的 4.2 节，这里主要讨论嵌入式微处理器的软硬核比较。

(1) 嵌入式微处理器软核是基于 LUT 的实现方式，电路的工作频率和功耗效率较差，并且会牺牲很多的 LUT 资源。FPGA 内部的嵌入式微处理器硬核的工作效率较高，但是最大的缺点就是对于不需要嵌入式微处理器的应用，这个硬核就是一种硬件资源的浪费，并且它的价格成本较高。

(2) 从指令集的角度来看，嵌入式微处理器硬核在芯片制造后就不能根据具体的应用更改或者更新指令集。对于嵌入式微处理器软核来说，一般都可以根据用户的应用添加定制指令。这些定制指令就是通过 FPGA 内部的 LUT 通用逻辑资源来实现的。嵌入式微处理器软核所提供的定制指令确实为通用处理器设计增加了一大亮点，使得通用灵活性和高效专用性得到一定程度的结合。

(3) 从灵活性的角度来说，嵌入式微处理器软核十分灵活，只要 FPGA 的硬件资源足够，一个 FPGA 内部可以搭建多处理器系统。但是嵌入式处理器硬核就只能局限于芯片内

部的硬核处理器资源。这在芯片制造后就已经确定，无法实现对处理器核数量的现场可编程性。

(4) 从软件工具的使用来说，嵌入式微处理器硬核和软核并没有什么差别。一般的软件工具都能支持硬核或软核，相应的软件库也都没有什么区别。例如我们无法从图 1-17 和图 1-18 所提供的软件代码来确认后端的微处理器是硬核还是软核，这是因为嵌入式微处理器硬核和对应软核的基本架构都是一样的。目前 FPGA 虽然能够配置高速缓冲区(cache)的数量，中断地址或者是复位地址，但是还不能配置不同的总线结构、寄存器数量等系统架构的处理器软核。

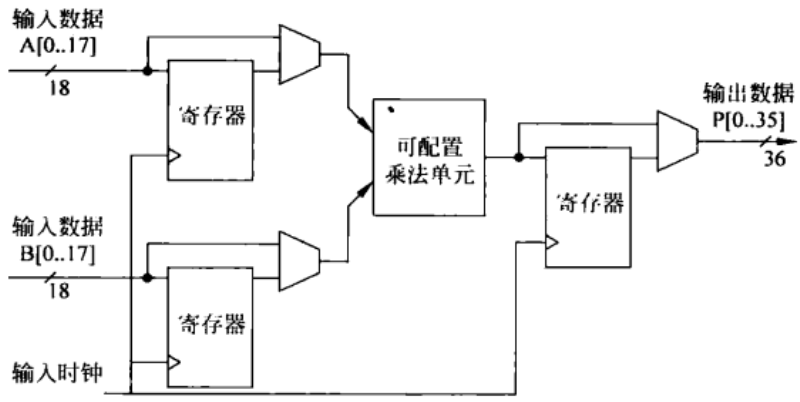
在 FPGA 内部嵌入微处理器软核或者硬核后，FPGA 的应用就从传统上的数字电路验证平台上升到了系统级设计平台，该系统能够提供基于通用处理器的软件编程和基于 LUT 的数字电路实现。基于 FPGA 的嵌入式系统目前已经非常普遍，具体介绍见第 4 章。现有的可重构系统很多都是利用系统级 FPGA 来实现的。可重构系统的介绍见第 6 章。

2.4.3 嵌入式 DSP 模块

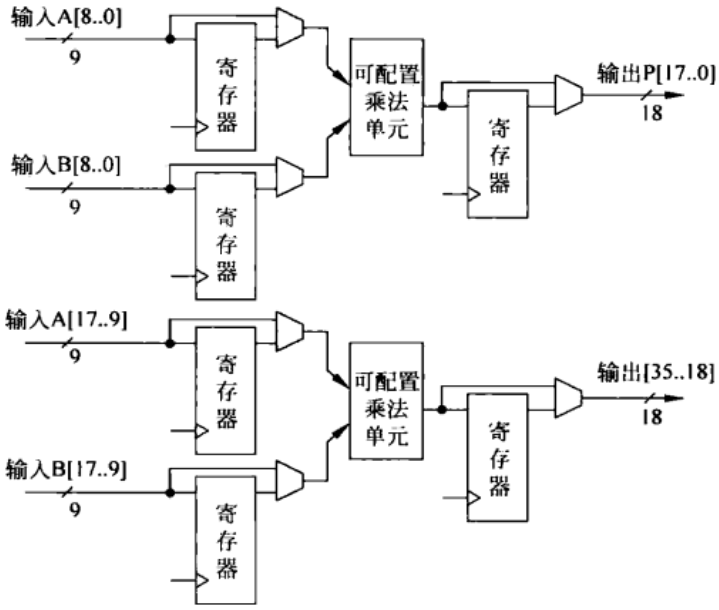
相对通用型微处理器和传统的通用型 FPGA 来说，DSP 处理器存在的优势主要是它能够实现快速并发的乘加运算和成熟的软件建模和仿真工具。经典的单核微处理器一个时钟只能实现一次乘加运算，基于 LUT 实现的处理器软核的乘加性能也不会很高，比不上 DSP 处理器中的乘法器和累加器的性能。即使对于滤波器中非常普遍的一个常数和另一个变量相乘的常数乘法器，用 LUT 或存储器也可以实现基于查找的乘法器，但是它们的性能还是无法和 DSP 处理器中的乘法器相比。因此，如果在 FPGA 内部直接嵌入多个 DSP 模块，甚至是 DSP 模块阵列，用于实现并行的乘加运算，那么这种异构的 FPGA 器件将具有比传统的 DSP 芯片更高的并行性。FPGA 所嵌入的 DSP 模块并没有很好的编译器或软件调试器，或者是类似于 Matlab 的系统功能仿真工具。虽然 FPGA 内部所嵌入的 DSP 乘加模块和 DSP 芯片相比使用起来非常不方便，但是并行化乘加器所带来的性能提升却是无法忽视的优点。例如 BDTi 公司[20]要把一个 DSP 基准测试例子从 Matlab 格式导入到 FPGA 器件运行，在商用 FPGA 供应商的密切配合下，一般都需要好几个月时间才能充分发挥 FPGA 所提供的性能优势。但是这样做的结果也是很有收获的。同一个应用在 FPGA 器件上的运行效率一般要比在 DSP 芯片快几十倍。

虽然 FPGA 内部所嵌入的 DSP 模块和 DSP 芯片所提供的工具链的成熟度相差很多，但是这些 DSP 模块具有硬件可编程所带来的灵活性，例如商用 FPGA 器件内部的 DSP 模块一般都可以有不同的配置模式，一个 $n\times n$ 位的乘加器可以作为两个 $n/2\times n/2$ 位的乘加器使用；或者把两个 $n\times n$ 位的乘加器合并为一个 $2n\times 2n$ 位的乘加器使用。这种灵活的配置模式非常类似于嵌入式存储器的工作方式，其优越性是传统的 DSP 芯片所无法提供的。

图 2-26(a)是支持符号数和无符号数的可配置乘法器示意图，它的输入和输出都是可以带寄存器和不带寄存器的，这样便于有效地提高乘法器的时序性能和接口的灵活性。它的输出可以送到下级的累加器，以实现乘加功能。根据用户对数据精度的需求，图 2-26(a)中的 18×18 位乘法器可以拆分为两个 9×9 位乘法器，这样便于更好地提高硬件资源利用率。在硬件实现上，由于乘法器一般是基于加法器来实现的阵列化结构，因此这种拆分或者合并的硬件共享方法是比较容易实现的。



(a) 支持符号数和无符号数的可配置乘法器



(b) 拆分后的符号数和无符号数的可配置乘法器

图 2-26　灵活配置的乘法器示意图

综合考虑 FPGA 内部所嵌入的通用性微处理核、存储器核、阵列化的 DSP 模块和已有的通用型数字电路实现功能已经极大地扩展了 FPGA 的应用领域和性能优势。

2.5　可编程逻辑单元结构比较

本章讨论了基于 AND/OR、MUX 和 LUT 的可编程逻辑单元，其中着重介绍了基于 LUT 的可编程逻辑器件结构。这一节我们基于一定的假设，定量分析 AND/OR、MUX 和 LUT 可编程逻辑结构的优缺点，从而加深对 LUT 结构优越性的理解。我们是基于以下假设进行结构性能比较的。

(1) 假设所采用的半导体工艺相同，即不考虑针对某些确定工艺对晶体管参数进行优化调整的情形。可编程单元面积是用可编程单元所需要的晶体管个数来衡量。

(2) 采用基于 SRAM 的硬件可编程技术,即一个编程点需要 6 个晶体管。

(3) 不考虑输入和输出可能需要的电平恢复单元,例如反相器等。这些单元所需的晶体管个数相对较少。我们这里主要考虑用于实现逻辑函数的晶体管单元数目及其随着输入变量数变化的趋势。

(4) 可编程单元的灵活度或者实现效率通过设置配置点的不同值能够实现的组合逻辑函数总数与所需要的晶体管个数之比来衡量。

根据以上假设我们先来计算图 2-1 所示基于 AND/OR 的 PLA 结构的实现效率。它有 n 个输入变量,经过反相器后共有 $2n$ 个信号输入到可编程与阵列。它有 p 个乘积项输出到可编程或阵列,可编程或阵列有 m 个组合函数输出。一般 p 和 m 都远小于 2^n。在可编程与阵列和或阵列中,每个交叉的地方不一定都有配置点。假设在所有的交叉点中,只有比例为 a 的交叉处有配置点,$0<\alpha\leqslant 1$。因此可编程与阵列和或阵列所需要的配置点数为

$$C_{\mathrm{PLA}} = \alpha(2np + pm) \tag{2-6}$$

每个 SRAM 配置点需要 6 个晶体管,因此配置点共需要的晶体管数目为

$$C_{\mathrm{PLA\_T}} = 6\alpha(2np + pm) \tag{2-7}$$

再来计算与或门所需要的晶体管数目。假设一个 i 位输入的与门和或门均需要 $2i$ 个晶体管($0\leqslant i\leqslant n$)。在可编程与阵列中,n 个输入 x_i 及其反相输出 $\bar{x}_i$ 共有 $2n$ 个输入线($0<i\leqslant n$)。一个乘积项可以是 1、x_i 或者 $\bar{x}_i$ 的乘积组合。由于 $x_i\,\bar{x}_i=0$,因此同一个变量的 x_i 及其反相输出 $\bar{x}_i$ 不能出现在同一个乘积项中。因此,对一个变量 x_i 来说,它出现在一个乘积项中的可能性有且只有三种:1、x_i 或者 $\bar{x}_i$,可以把它们统称为"文字"或简称"字"。因此 n 个变量共能组成 3^n 个有效乘积项。例如以 $n=3$ 为例,共能产生以下 27 个乘积项。要满足可编程逻辑的完备性,这些乘积项都需要产生。

$$1,x_0,\bar{x}_0,x_1,x_1x_0,x_1\,\bar{x}_0,\bar{x}_1,\bar{x}_1,x_0,\bar{x}_1\,\bar{x}_0,\bar{x}_1\,\bar{x}_0,x_2,\cdots,\bar{x}_2\,\bar{x}_1\,\bar{x}_0 \tag{2-8}$$

如果变量 x_i 在一个乘积项中的三种取值 1、x_i 或者 $\bar{x}_i$ 分别用 0、1、2 来表示,那么式(2-8)中三变量的所有乘积项都可以用以下编码来表示。它共有 27 种可能性:

$$000,001,002,010,011,012,020,021,022,100,\cdots,222 \tag{2-9}$$

在式(2-9)编码的乘积项中,"0"表示对应的变量不在乘积项中出现,因此这个变量就不需要连到与门。"1"和"2"分别对应于 x_i 和 $\bar{x}_i$,它们都需要连到与门作为输入,于是 n 个变量共有 $2n$ 个输入要连到与门产生可编程与阵列的一个输出,它所需要的晶体管数为 $4n$。由于可编程与阵列共产生 p 个乘积项,或者说 p 个与门,总共需要 $4np$ 个晶体管。

对于可编程或阵列来说,它共有 p 个输入,m 个输出,其中每一个输出都对应于一个函数的实现,于是可编程或阵列中 m 个或门所需要的晶体管数目为 $2pm$。因此可编程阵列所需要的晶体管数目为

$$P_{\mathrm{PLA\_T}} = 4np + 2pm \tag{2-10}$$

对于 PLA 结构所能实现的函数个数,是比较难以准确计算的问题。我们先计算 m 个输出中的一个输出所能实现的函数表达式的个数。注意这里函数表达式和函数是不同的两个概念。同一个函数经过不同的化简方法就可以得到多个表达式,因此函数表达式的数目要多于函数的数目。我们已经知道可编程与阵列中的 p 个输出中的每一个都有 3^n 个可能的乘积项输出,在这 p 个乘积项中可以选择 j 个乘积项构成一个有效的函数表达式($0\leqslant j\leqslant p$)。

换句话说，如果一个函数表达式只包含有一个乘积项，那么它有 $C_{3^n}^1$ 种可能；如果一个函数表达式只包含有两个乘积项，那么它有 $C_{3^n}^2$ 种可能。依此类推，对于有 p 个输入的可编程或阵列来说，它所能实现的函数表达式个数为

$$C_{3^n}^0 + C_{3^n}^1 + C_{3^n}^2 + \cdots + C_{3^n}^p = \sum_{i=0}^{p} C_{3^n}^i \tag{2-11}$$

以三变量式(2-8)中的27个乘积项为例，假设 p 值为4，即最多可以有4个乘积项组成函数输出。如果表达式中只能包含一个乘积项，那么这些27个乘积项就有27个函数表达式，它们彼此各不相同。如果表达式中有两个乘积项，那么在物理上只要连通不同的编程开关就能够实现的函数表达式个数为 $C_{3^3}^2 = 351$ 个。例如 $x_0 + x_1$，$x_1 + \bar{x}_1 x_0$，$x_0 + x_1 \bar{x}_0$ 这三个函数表达式分别对应于三种配置方式，但是它们实际上都实现了同一个函数 $x_0 + \bar{x}_1$。因为第二种配置所对应的函数表达式 $x_1 + \bar{x}_1 x_0$ 和第三种配置所对应的函数表达式 $x_0 + x_1 \bar{x}_0$ 都可以化简为 $x_0 + x_1$，因此虽然我们可以通过配置不同的开关产生不同的函数表达式，但是这些表达式大多是没有优化过的，或者说是冗余的。由此可见，这些在物理上可以通过可编程或阵列中的编程开关实现的函数表达式有很大的冗余。虽然物理上所实现的函数表达式不同，但是其中很大一部分函数表达式都是冗余的，即它们可能对应于同一个逻辑函数。例如以三变量函数 $n = 3$，$p = 4$ 为例，由式(2-11)可以算得能够实现的函数表达式数量为 $\sum_{i=0}^{4} C_{3^3}^i = 1 + 27 + 351 + 2925 + 17550 = 20854$。而三变量函数一共才有 $2^{2^3} = 256$ 个，这说明在20854个函数表达式中有98%以上是冗余的，它们只是用不同的表达式实现了同一个函数而已。这也可以说明可编程或阵列中大部分可编程开关是冗余的。如何选择可编程开关的分配方式，使得这种结构能够实现所有或者是大部分的布尔函数，这是一个比较复杂的优化问题。在最优的情况下，我们假设它能实现所有的 n 变量函数，即共有 2^{2^n} 个。由于可编程或阵列共有 m 个输出，出于对驱动能力等因素的考虑，我们假设这些 m 个输出相同时也视为有效输出。因此 m 个输出的 PLA 所能实现的函数个数为

$$F_{\text{PLA}} = m \times 2^{2^n} \tag{2-12}$$

则 PLA 的最佳实现效率为

$$E_{\text{PLA\_Best}} = \frac{m \times 2^{2^n}}{C_{\text{PLA\_T}} + P_{\text{PLA\_T}}} = \frac{m \times 2^{2^n}}{6\alpha(2np + pm) + 4np + 2pm}$$

$$= \frac{m \times 2^{2^n}}{(12\alpha + 4)np + (6\alpha + 2)pm} \tag{2-13}$$

对于 MUX 来说，如图 2-9 所示，由于它所有的数据端 x_i 和选择端 s_j 均作为用户输入端，而不是连接于配置点，因此它并不具有我们所要求的现场可编程性——通过配置点实现不同的功能。如果把它的数据端 x_i 连到配置点，选择输入端 s_j 作为用户输入，那么它就成为了一个 LUT。如果把它的选择端 s_j 连到配置点，它的数据端 x_i 作为用户输入，那么它就适合于互连线的连接控制了。目前还没有发现把 MUX 中的部分数据端和部分选择端连到配置点的组合方式，或许这种配置方式能够在可编程逻辑单元和可编程逻辑资源之间实现动态切换。因此，对于 MUX 作为可编程逻辑单元实现来说，我们把它退化为 LUT 看待。

对于有 n 个输入端的 LUT 来说，它所需要的配置点个数为

$$C_{\text{LUT}} = 2^n \tag{2-14}$$

根据图2-16中LUT的晶体管电路，我们可以知道n个输入的LUT从输入到输出共有$2n$级，分别由n个变量和它们的反相器输出来控制。最初的二级共需要$2\times2^{n-1}=2^n$个传输管；后续两级需要$2\times2^{n-2}=2^{n-1}$个传输管。依此类推，最后两级共需要传输管个数为$2\times2^{n-n}=2$。因此n个输入的LUT共需要传输管的个数为

$$P_{\mathrm{LUT}}=2^n+2^{n-1}+\cdots+2=\sum_{i=1}^{n}2^i=2^{n+1}-2 \tag{2-15}$$

根据2.3.1节的讨论，n输入LUT能够实现所有的n输入函数，共有2^{2^n}个，即

$$F_{\mathrm{LUT}}=2^{2^n} \tag{2-16}$$

因此LUT的实现效率为

$$E_{\mathrm{LUT}}=\frac{2^{2^n}}{2^n\times6+2^{n+1}-2}=\frac{2^{2^n}}{2^{n+3}-2} \tag{2-17}$$

比较式(2-13)和式(2-17)的实现效率，即单个晶体管所能实现的单输出函数个数，我们可以发现PLA结构在α为0.3时就能假设实现所有函数的前提下，它所需要的晶体管个数要远远多于LUT需要的晶体管数目，即两式中的分母值相差很大。由图2-27的曲线就可以看出这种成本差别。另外从硬件结构上可以看到，用于可编程与或阵列中的晶体管相互之间会影响，冗余度较大。这也可以说明PLA的改进版本，即保留可编程的与阵列、改用固定连接的或阵列结构PAL反而比PLA得到更广泛的应用。PAL结构比PLA结构成本低、延时小，函数实现的灵活性可以通过可编程与阵列和软件算法进行优化。实际上最初提出PAL思想的是UC Berkeley从事Espresso的研究人员[2]，他们觉得可以利用软件的优势对PLA结构进行简化，从而得到了PAL结构。和PLA或者PAL结构相比，图2-16所示LUT的晶体管和配置点结构非常简单清楚。配置点用于存储函数的真值表，传输管用于打通配置点和输出的通路，不同的变量输入组合就打通对应的通路，这些通路之间彼此不会影响，因此基于可编程与或阵列的逻辑结构基本上就完全被基于LUT的结构所取代。

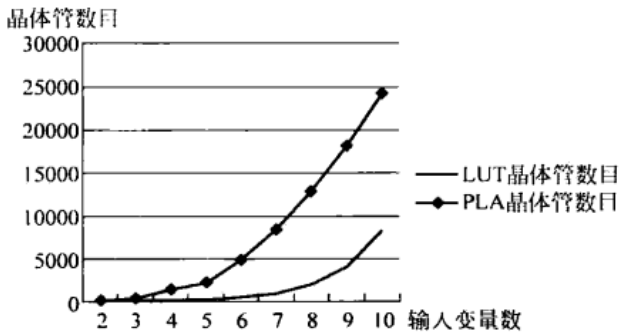


图2-27　实现单输出函数PLA和LUT所需要的晶体管数量比较

综上所述，基于可编程与或阵列的PLA或者PAL结构的性能比不上LUT单元，尽管两者都具有完备性。相对应的软件算法也是前者比后者复杂很多。前者对应的二级电路优化工具Espresso[2]虽然非常成熟，但是不能保证对所有的函数都能找到最优结果。而LUT结果只需要知道函数的真值表就可以了，只要变量数满足，就能够实现任意的函数，因此软件实现十分方便。MUX单元功能十分强大，它可以作为可编程逻辑单元使用，也可以作为

可编程互连资源使用。当 MUX 的选择端数较少时，它们就可以作为变量输入。而数据端就可以接到配置值，这就实现了 LUT 的功能。因此可编程逻辑单元中的 MUX 选择端数据都比较小，一般不超过 3。当 MUX 的选择端数较多时，它所控制的数据端数就是它的幂值。因此选择端可以接配置点，数据端接较多的互连线段，这样就可以控制数目较多的互连线段的连接方式。因此作为可编程互连资源使用的 MUX 单元一般都比较大，所控制的互连端数目一般都超过 10。

习题

1. 基于图 2-3 中的 PROM 结构，假设电源电压为 5V，二极管和熔丝的导通压降为 0.8V，计算 R_1 和 R_2 的取值，即负载的限制，使得高电平输出时能够达到超过 2.4V 的 TTL 电平标准。

2. 根据图 2-5 的描述方式，写出图 2-28 中两个组合逻辑函数的表达式。

3. 说明 MUX 和 LUT 在实现逻辑函数时哪些输入是接到配置码点，哪些是接到用户的输入输出。

4. 如图 2-14 所示，与非门的输入输出端增加四选一 MUX 就可以实现现场可编程性。还有一种更简单的方式是在二输入与非门的输入和输出端之间添加可配置的二选一 MUX，如图 2-29 所示。每个 MUX 的选择端都用于控制与非门的输入输出是否为反相。这种单元共有两个输入端 A 和 B，一个输出端 f，三个配置端(图中未画出)。试计算这种可编程组合逻辑单元通过不同的配置所能实现的二输入函数个数。

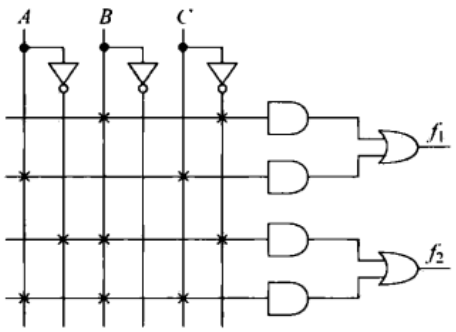


图 2-28　利用 PAL 结构实现组合电路示例

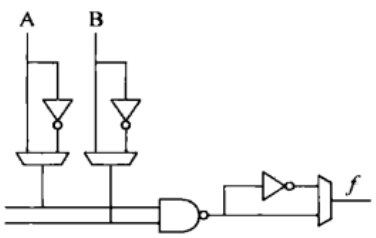


图 2-29　基于与非门的可编程组合逻辑单元

5. 统计图 2-15 基于 MUX 的单元所能实现的函数个数分布。

6. 把图 2-16 中的 LUT3 的输入变量次序从上到下由 *cba* 换成 *abc* 后，标出 SRAM 值从上到下的配置值。

7. 仿照式(2-4)或者式(2-5)用数学公式证明在表 2-2 中的“右边→下边”和“下边→右边”的连接关系在连接方向上是互逆的，即如果右边标号为 x 的互连线到下边标号为 y 的互连线存在连接关系，那么下边标号为 y 的互连线也一定能够连接到右边标号为 x 的互连线。

8. 根据图 2-20 所示分离型和 Wilton 开关盒结构的连接方式，对于可编程逻辑单元阵列数目为 $n\times n$ 的 FPGA 结构来说，假设互连资源只包含四倍线一条互连线长度横跨四个

可编程单元，其通道宽度为 W（不考虑和 IO 连接的开关盒），计算开关盒的数目；然后根据这两种开关盒的拓扑结构的不同，求得分离型开关盒比 Wilton 开关盒所能节约的传输管数目。注意晶体管的数目要包含实现每个 SRAM 配置单元所需要的 6 个晶体管。

9. 指出 PLA、PAL、LUT 和 MUX 可编程逻辑单元的优缺点。计算图 2-1 所示 PLA 结构所能实现的函数个数。

10. 假设 LUT 的码点值（表示函数的真值表）是可以动态改变的，即 SRAM 部分可以在 FPGA 运行时允许写入新的数据。试用一个包含一组 LUT4 和 DFF 的基本单元来设计四位计数器的时序电路。然后假设 LUT 的码点在电路运行时不能动态改变，那么画出基于包含四组 LUT4 和 DFF 的可编程逻辑簇的四位计数器电路实现。

参考文献

[1] Actel Corporation. Actel SX-A Family FPGAs Datasheet v5.3, February 2007

[2] Brayton R K, Hachtel G D, McMullen C T, Sangiovanni-Vincentelli A L. Logic Minimization Algorithms for VLSI Synthesis. Boston Kluwer Academic Publishers, 1984

[3] 黄均鼐，俞承芳，蒋慧文，卢煊. 可编程逻辑器件设计. 上海：复旦大学出版社，1997

[4] 宋立国，姜玉宪. 粗粒度可配置计算结构的研究与进展. 电子技术应用，2005，31(1)：19-21

[5] Brian Dipert. Vital Stats: Third Annual Programming-Logic Directory. EDN (www.edn.com), September, 2002: 45-64

[6] R. Ramanarayanan etc. Modeling Soft Errors at the Device and Logic Levels for Combinational Circuits. IEEE Transactions on Dependable and Secure Computing, 2009, 6(3): 202-216

[7] Lin M J. The Amorphous FPGA Architecture. Proceedings of the 16th International ACM/SIGDA Symposium on Field Programmable Gate Arrays (FPGA2008), February, 2008: 191-200

[8] Betz V, Rose J, Marguardt A. 著. 王伶俐，杨萌，周学功译. 深亚微米 FPGA 结构与 CAD 设计. 北京：电子工业出版社，2008

[9] Ma K J, Wang L L, Zhou, X G, et al. General Switch Box Modeling and Optimization for FPGA Routing Architectures. International Conference on Field-Programmable Technology (FPT2010), 2010: 320-323

[10] Clive Maxfield. The Design Warrior's Guide to FPGAs, Chapter 4, Newnes Elsevier, Oxford, 2004

[11] Ian Kuon, Russell Tessier, Jonathan Rose. FPGA Architecture: Survey and Challenges. Foundations and Trends in Electronic Design Automation, 2007, 2(2): 135-253

[12] Uwe Meyer-Baese. Digital Signal Processing with Field Programmable Gate Arrays (Third Edition). Springer, 2007: 3-4

[13] El Gamal A, Greene J, Reyneri J, et al. An Architecture for Electrically Configurable Gate Arrays. IEEE Journal of Solid-State Circuits, 1989, 24(2): 394-398

[14] Jonathan Rose, Robert J Francis, David Lewis, Paul Chow. Architecture of Field-Programmable Gate Arrays: The Effect of Logic Block Functionality on Area Efficiency. IEEE Journal of Solid-State Circuits, 1990, 25(5): 1217-1225

[15] Lemieux G G, Brown S D. A Detailed Router for Allocating Wire Segments in Field-Programmable Gate Arrays. Proceedings of the ACM/SIGDA Physical Design Workshop, Lake Arrowhead, CA, April 1993: 215-226

[16] Chang Y W, Wong D F, Wong C K. Universal Switch Modules for FPGA Design, ACM Trans. Design Automation of Electronic Systems, 1996, 1(1): 80-101

[17] Wilton S. Architecture and Algorithms for Field-Programmable Gate Arrays with Embedded Memory. Ph. D. thesis, University of Toronto, 1997

[18] Tsu W, Macy K, Joshi A, et al. HSRA: High-Speed, Hierarchical Synchronous Reconfigurable Array. ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, 1999: 125-134

[19] Pande P P. Performance Evaluation and Design Trade-Offs for Network-on-Chip Interconnect Architectures. IEEE Transactions on Computers, 2005, 54(8): 1025-1040

[20] http://www.bdti.com/

第3章

基于FPGA的数字电路设计

3.1 高级描述语言编译和芯片版图生成流程

作为通用的软件计算和硬件设计平台，基于通用处理器的软件编译和基于FPGA的硬件设计过程有很多相似之处。早期简单的软件程序，可以直接采用汇编语言来编写。随着程序规模的扩大以及对源代码的可读性、可维护性和可重用性的需求，我们必须借助于软件编译器来自动实现从高级语言描述到汇编语言和二进制可执行程序的转换过程。软件编译器就是把用户用高级语言描述的源代码翻译成目标处理器上可以运行的二进制可执行程序。对于FPGA编译器来说，它也是把用户输入的RTL描述源代码翻译成可以下载到FPGA器件的位流文件，从而配置成用户所需要的电路的实现方式。因此，我们先讨论这两种通用计算方式的特点，然后基于熟悉的高级语言软件编译流程来理解FPGA的位流文件产生过程。

3.1.1 基于通用处理器的软件编译流程

经典的通用型处理器和FPGA都具有现场可编程性，因此它们都属于通用的计算平台，但是两者的计算效率是不同的。对于经典的通用处理器来说，计算程序的复杂度主要表现在程序所需要的存储器空间和运行时间。对于给定的硬件资源，包含微处理器和存储器，只要它们能够一直保持供电状态，那么从理论上来说所能运行程序的复杂度主要受机器运行寿命的限制，即越复杂的计算则需要更多的运行时间。为了提高计算效率，从而减少计算程序的运行时间，通用处理器应采取越来越复杂的流水线结构和更高的时钟频率。存储器的容量也急剧扩大，供微处理器在更短的时间内访问更多的数据。近年来由于电路工作的时钟频率已经逐渐趋近物理极限，并且电路的功耗成为比时序性能更优先考虑的因素，通用处理器开始采用多核结构，即增加硬件上的并行性(parallelism)和并发性(concurrency)来提高计算效率。如多核CPU和GPU都是典型硬件并行化的例子。另一方面，对于通用的FPGA计算平台来说，器件内部的各种硬件资源，都是可以并行工作的。相对于通用处理器来说，虽然FPGA要比通用微处理器难用很多，并且用户还需要掌握硬件设计的基本知识，但是同一个计算任务一般在FPGA上的计算效率要比通用微处理器高几十倍左右，尽管FPGA电路的最高时钟频率一般要远远低于通用处理器的最高工作频率。由于硬件资源规模的有限性，即一个FPGA器件不可能具有无限多的可编程逻辑资源，有些计算任务在给定的FPGA器件上就无法运行。因此，为了进一步提高计算能力，一块FPGA芯片上具有的芯片规模基本上是按照指数规律增大。这种硬件资源规模增大的

速度和2004年之前处理器时钟频率的提高速度基本上都遵循摩尔定律。另外，系统级FPGA芯片还要嵌入更多的专用硬件资源，用于进一步提高粗粒度计算的效率和晶体管的资源利用率。

因此，从可计算性和计算效率的角度来看，通用处理器和FPGA所具有的可编程性的性能有以下区别。

(1) 对于通用处理器来说，单核处理器的计算任务大体上还是基于串行的工作方式，即在同一时刻处理器按照串行的方式执行指令，因此一个复杂的计算任务相对就要耗费较多的计算时间。通用处理器硬件资源本身除了器件使用寿命外，并没有对运行时间提供其他限制。但是对于FPGA来说，计算任务是在各硬件资源上并行执行的，计算任务的可计算性主要取决于器件的硬件资源规模。这就是说，FPGA的硬件资源规模基本上就约束了计算任务的复杂度。第6章所讨论的可重构方法是在时间上重新利用FPGA的可编程资源。这种方法为突破FPGA硬件资源规模的限制提供了有效途径。对于静态可编程的非可重构FPGA来说，硬件资源规模束缚了计算任务的规模。

(2) 对于通用处理器来说，计算效率可以用在给定的硬件资源及其计算性能下计算程序所需要的运行时间来衡量。计算效率的优化主要是利用软件的方法进行。FPGA的计算效率主要取决于在确定器件下电路工作的最高时钟速度。

对于单核通用微处理器来说，相应的软件编译和运行环境已经非常成熟。如GCC (GNU compiler collection，GNU套装编译器)工具和Linux操作系统就能支持多种CPU结构。但是对于通用的FPGA器件来说，不管编译工具还是运行环境，目前远没有达到由统一的后端布局布线工具支持不同公司FPGA器件的水平。对于高级语言编译器来说，一般分为前端分析和后端综合两大部分[1]。例如，对于常用的GCC工具来说，它的前端部分支持多种语言：C、C++、Java、Fortran、Ada；后端综合部分支持多种处理器，例如Intel系列处理器、ARM、PowerPC等处理器结构。如果今后需要支持一种新语言，只要在前端分析部分增加对这种语言的前端分析即可，后端综合部分都是可以共享的。另一方面，如果需要支持一种新型处理器的编译功能，只需要在后端综合部分添加即可，前端分析部分完全可以共享。

如图3-1所示，编译器首先读取源程序代码，进行词法、语法分析。词法分析就是把代码中的各个字符串转换为单词，包括高级语言的关键字和用户定义的标识符等。语法分析就是把经过词法分析后的各个单词按照高级语言所定义的语法规则构成合法的树结构。如果输入源程序没有不认识的单词和语法错误，那么编译器就由输入源程序得到对应的语法树结构，再经语义分析，确定源程序的含义。经过语义分析后，语法分析所生成的树结构中的节点就赋予了程序的语义信息，即编译器理解了程序的行为和功能。如果源程序没有语义错误，那么就可以生成中间代码的表达方式。以我们日常所用的自然语言为例，“学生”、“是”、“我”都是合法的单词，但是这些单词的不同组合，“我学生是”是有语法错误的表达方式，“我是学生”才能正确地表达语义。一般来说，这种正确表达语义的中间格式(intermediate representation)是以RTL(register transfer language，寄存传输语言)来描述。对于RTL语言的具体格式定义，可以参考GCC的在线文档[2]或GCC参考文献[3]。对于GCC工具来说，命令行“gcc -dr 输入文件名”就可以产生对指定输入文件转换得到的RTL中间输出文件，其中命令行参数“-dr”表示输出RTL的调试(debug)信息，即分别取自RTL和debug的首字母。默认情况下，输出的文件扩展名为.rtl。

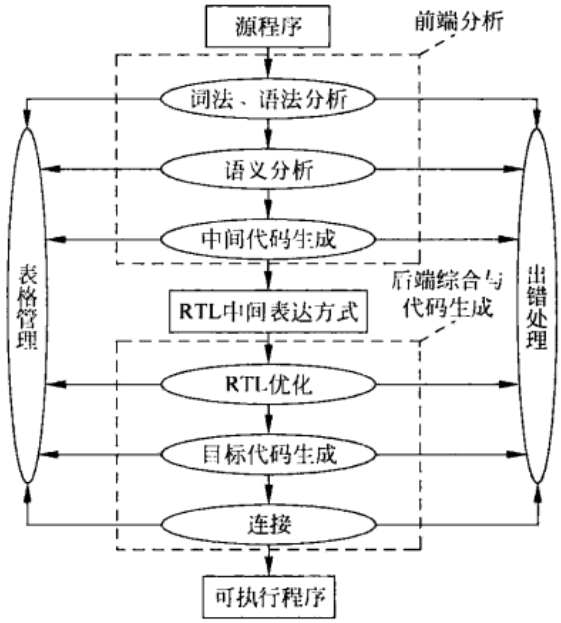


图 3-1　高级语言编译流程[1]

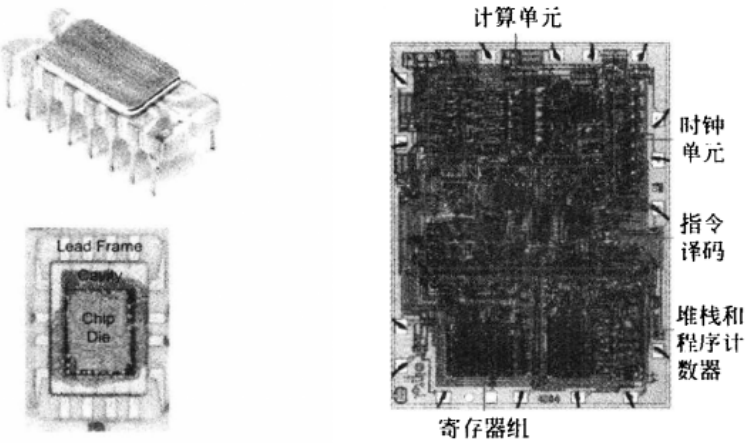
源程序经过前端分析得到 RTL 中间表达方式后，后端综合阶段就基于这种 RTL 中间表达格式和处理器的硬件结构信息，进行 RTL 优化，并生成能够在目标处理器上运行的二进制代码。RTL 优化过程包含和处理器无关的树结构优化，以及和具体处理器结构相关的优化。一般来说，输入源程序会调用一些库函数，或者大规模源程序一般由多个源文件来描述。因此多个源文件所转换到的目标(object)文件就需要连接成一个可执行(executable)程序，从而可以确定各个目标文件之间的函数调用关系及其函数调用所需要的首地址。对于 C/C++语言来说，这个可执行目标程序只包含一个 main 函数。在处理器运行时它就可以从 main 函数被装载到内存后的首地址开始运行指令。如果程序是在操作系统的管理下运行的，那么在用户程序的 main 函数前一般还会连接操作系统提供的引导程序。因此，虽然 C/C++等语言规范是和操作系统无关的，但是编译器一般和操作系统相关。同一个编译器在不同的操作系统下也需要提供不同的函数库，以保证用户程序的正常运行。

在高级语言的编译过程中，还需要从源程序中提取并保留各个变量的名字、用户定义类型的名字、函数的名字、参数个数和返回类型等函数原型信息。编译程序就把这些名字和类型等信息记录到符号表中，方便各源文件的查找调用，或者在程序连接时验证检查，抑或是在调试程序时方便找到声明或引用某符号的行号等信息。在编译过程中符号表的查找是非常频繁的，因此就需要编译程序能够有效地管理这些符号表格。这就是图 3-1 中的“表格管理”功能。

在编译流程中各个功能模块出现的各种错误，例如前端分析流程中词法、语法分析模块检测到单词组成错误、括号不匹配或者其他语法错误，语义分析模块发现某运算符和它的运算对象类型不符等都需要输出合适的错误提示信息，帮助用户有效地修改源程序代码。因此图 3-1 中的“出错处理”需要判断并产生合适的诊断、警告和错误等信息，帮助用户快速定位代码中的错误和可能存在的危险，提高用户编写程序的效率。

3.1.2　基于 EDA 工具的数字电路设计流程

随着通用计算机软硬件技术的不断发展，它的应用领域不断扩大，逐渐渗透到我们生活和工作的各个领域，从而改变了我们的生存和生活方式。一开始数字电路基本上是基于晶体管，利用笔和纸的方法进行手工设计，这就是早期所谓的晶体管-晶体管逻辑(TTL)电路。例如 1971 年 11 月 Intel 公司发行的世界第一块通用处理器 4004，位宽为 4，能够满足当时计算器的功能需求。图 3-2(a)所示为双排 16 管脚封装的 4004 芯片。去除封装材料后，可以清楚地看出芯片裸片四边各有 4 个引脚。图 3-2(b)所示为芯片内部的版图，它共集成了 2300 个晶体管。随着芯片集成度和规模的扩大，一般超大规模集成电路会集成高达上亿个晶体管，因此基于晶体管的手工设计方法已经无法满足需求。这就好像只用汇编语言无法编写大规模的软件程序一样，数字电路的设计需要计算机辅助设计方法，这样才能设计更高集成度和更大规模的芯片。硬件电路设计水平的持续发展，也有利于设计出更高性能的通用处理器，从而提高了软件程序的运行效率，另外也对电子设计自动化软件提出更高的需求。因此，硬件电路和设计自动化软件两者相辅相成，是集成电路在过去近半个世纪以来基本上按照摩尔定律所预测的那样呈指数发展的一个重要基础。



(a) 封装后的芯片和去除封装材料的裸片　　(b) 芯片内部版图

图 3-2　Intel 四位通用处理器芯片 4004 及其版图[4]

从当前数字集成电路产品的角度来看，主流的方法包括芯片设计和芯片制造两个基本独立的步骤。这种设计和制造的分工可以提高芯片的生产效率并能降低成本，因此芯片设计者的主要目标是如何设计出制造芯片所需要的电路版图，它包含了电路中所有的晶体管位置、大小和相互之间复杂的连接关系。GDSII 文件格式是芯片版图信息描述的工业界标准，芯片制造商可以根据 GDSII 文件提供的信息恢复版图形状并制造出所需要的芯片裸片，再经过封装、测试就可以作为合格的产品销售。如第 1 章所述，集成电路产品主要经过芯片设计、芯片制造和封装测试三个技术环节。本节主要讨论的是芯片设计技术，它又可分为前端综合设计和后端布图设计两个部分，如图 3-3 所示。前端综合设计主要是读入电路的 RTL 描述，或者是电路原理图的描述，由逻辑综合 EDA 工具输出针对一个半导体工艺

库优化的门级网表。后端布图设计主要是输入前端产生的门级网表，确定版图布局规划(floorplan)，由布局布线 EDA 工具输出针对同一个工艺库的版图文件。为了保证布图后的电路和前端综合后的网表具有相同的功能并且达到预期的时序性能，还要对布局布线后的电路运行后端时序仿真、静态时序分析和其他的验证过程。另外考虑到芯片功耗、热点(hot spot)对电路性能和寿命的影响，深亚微米集成电路还需要运行功耗分析和热分析等 EDA 工具。最后在产生 GDSII 文件前，对芯片版图做设计规则检查(design rule check，DRC)，确保芯片版图满足指定工艺的电气和物理要求，例如电源线的驱动、金属线的宽度和间隔要求等。只有布图软件产生的版图通过了后端时序分析、功耗分析、LVS(layout versus schematics)版图验证、DRC 检查后，所产生的 GDSII 版图文件才可以递交给芯片制造商生产。由于 FPGA 和专用芯片的后端设计目的、工具链和基本方法差别较大，因此本书主要介绍专用芯片的前端设计过程。它和 FPGA 的前端设计比较类似。对于专用芯片的后端设计流程，有兴趣的读者可以参考文献[5]。

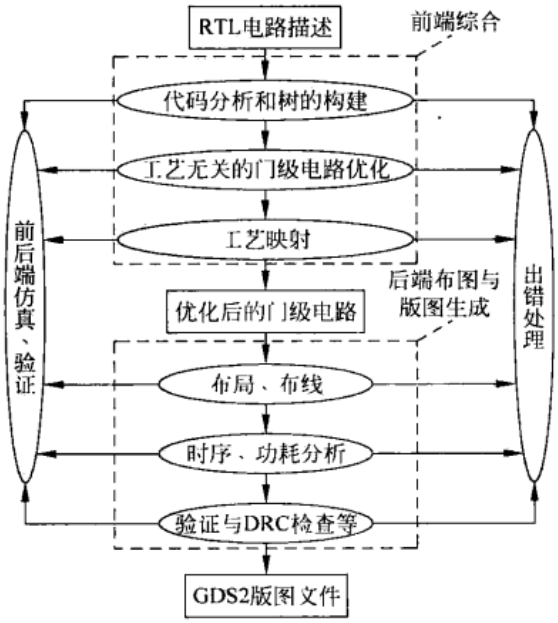


图 3-3 基于 EDA 工具的数字集成电路设计流程

当数字电路硬件工程师需要设计一款芯片时，一般采用硬件描述语言(HDL)来描述所需要的电路功能，然后由各种 EDA 设计工具进行仿真、综合和布图得到芯片制造所需要的版图文件。以 Verilog 硬件描述语言为例，一般在电路设计过程中包含以下三个层次[6]。

(1) 行为级或算法级：这是 Verilog 语言最高的抽象级别。相比较而言，VHDL 语言在行为级描述方面比 Verilog 语言具有更强大的功能。但是 Verilog 仿真器的速度一般要比 VHDL 仿真器快很多。在国内 Verilog 语言比 VHDL 语言的使用普及率要高。这个级别所描述的主要是算法行为，而不关心具体的硬件实现。这种描述方式和 C 语言编程很类似。各种循环语句，例如“while”、“for”和各种算术逻辑运算符是行为级描述的典型用法。这个层次有时也称为系统级描述。

(2) 数据通路级：设计者先设定基本的数据通路(data path)结构。数据通路确定了触发器和寄存器的数据传输方式或基本的流水线结构。为了实现数据在各个触发器或寄存器之间的变换和传输，就需要和数据通路配合的控制流程(control flow)。在这个层次的描述中就包含了时钟节拍及其寄存器的时序信息。因此这个层次的代码经常被称为 RTL (register-transfer level，寄存传输级)描述。

(3) 门电路级：电路描述中的各个模块直接确定了逻辑门(不是逻辑运算符)及其相互之间的连接关系。常用的逻辑门有 and、or、xor 等。用门级描述的电路实际上就是硬件电路的实现，它需要设计者已经熟悉硬件电路的构建。相对算法级和数据通路级来说，门级电路描述是偏向硬件层次的底层描述方式。

一般来说，高层次的描述方式灵活性较大，但是所得到的电路性能相对较差。除非用户对硬件电路设计非常熟悉，一般我们不采用门级描述来实现数字电路。对于 Verilog 语言来说，以上三个层次的描述也不是非常严格，并且它的行为级描述功能不如 VHDL 硬件描述语言强大。在混合采用以上三种描述层次时，我们有时也把行为级和数据通路级的混合描述方式统称为 RTL 描述，相应的电路结构如图 3-4 所示。这个图和第 2 章的图 2-7(a)是一致的。输入数据经过各组合电路的变换后在寄存器之间传输，直到输出所需要的变换结果。当输入数据只有一位时，那么图中的存储单元就是一个触发器。寄存器一般表示多位数据的存储，例如 8 位、16 位、32 位存储器等。多位寄存器之间的数据传输就通过一组连线连接。在 RTL 描述的电路中，输入数据在同步时钟信号的作用下，经过多级的组合逻辑电路转换。中间数据转换结果放到寄存器中，最后的转换结果由输出端得到。当然，输入和输出也可以增加寄存器，可以保证和前后级电路的同步。对于复杂的数字电路来说，可能包含多个时钟和多种连线或者总线结构。目前主流的工业标准 RTL 硬件描述语言是 Verilog 和 VHDL。Verilog 语言借助了很多 C 语言中的语法和语义，入门比较容易。VHDL 语言在语法和语义上的要求要比 Verilog 更为严格，比较适合于更高层次的系统级电路描述。本书中采用 Verilog 语言作为示例。

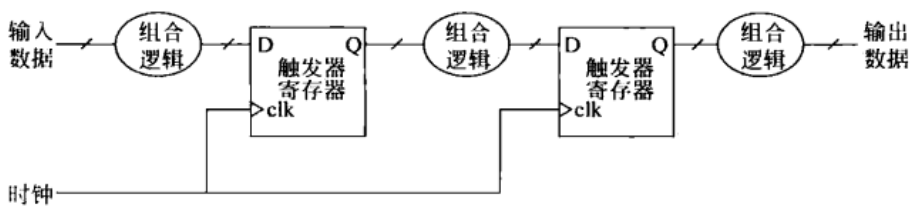


图 3-4　RTL 语言描述的数字电路基本结构

在图 3-3 的流程中，逻辑综合 EDA 工具首先读入用户的 RTL 描述文件，进行语法语义分析。如果代码分析没有错误，那么就进行电路结构的构建(elaboration)，得到类似于图 3-4所示的数据通路或者控制逻辑电路结构。一般来说在编译程序内部电路结构是用树的形式来表达，其中树的节点表示各种逻辑和算术运算，树的边表示各种运算之间的输入输出关系。在构建 RTL 电路结构时，主要是基于模板(template)的方式把 RTL 代码中的各种描述结构转换成逻辑电路，即编译器会识别代码中的关键字和一些固定的描述模板，从而生成优化的电路结构。

图 3-5 给出了三种基于 Verilog 语言的电路描述 RTL 模板。如果图中各模块的输入输

出不只是一位，而是多位的总线或者是数组形式，那么转换后的电路就是一位电路的重复复制。对于时序部分，边沿触发的触发器一般通过“always @(posedge 信号名)”来识别，如果数据位宽不只是一位，那么综合工具就会把它转为寄存器结构，其中触发器或者寄存器的输入信号可以由复杂的组合逻辑电路产生。最常用的组合电路可以通过 assign 语句来识别。在图 3-5 中，assign 语句可以由等价于 C 语言中的问号表达式、if-else、case 或者逻辑运算符来实现，它们都会生成右边的数据选择器或者更一般的逻辑电路。一般来说，右边的逻辑电路都是由简单的基本逻辑门构成。如果要利用一些已经优化的 IP 核，那么在代码分析时就要识别这些 IP 核，避免使用基本门电路来实现这些 IP 核的功能。这些组合逻辑电路可以通过输入输出级联进一步构建更复杂的组合电路。同时复杂组合电路的输出又可以输入到寄存器的输入端，从而构建出复杂的时序电路，满足系统应用的需求。对于 Verilog 和 VHDL 语言，两大国际标准 IEEE 1364.1 和 IEEE 1076.6 分别定义了可综合的 RTL 语言模板。通过标准中所定义的模板，逻辑综合工具在构建的时候就可以产生初始的门级电路结构，供后续作电路优化。

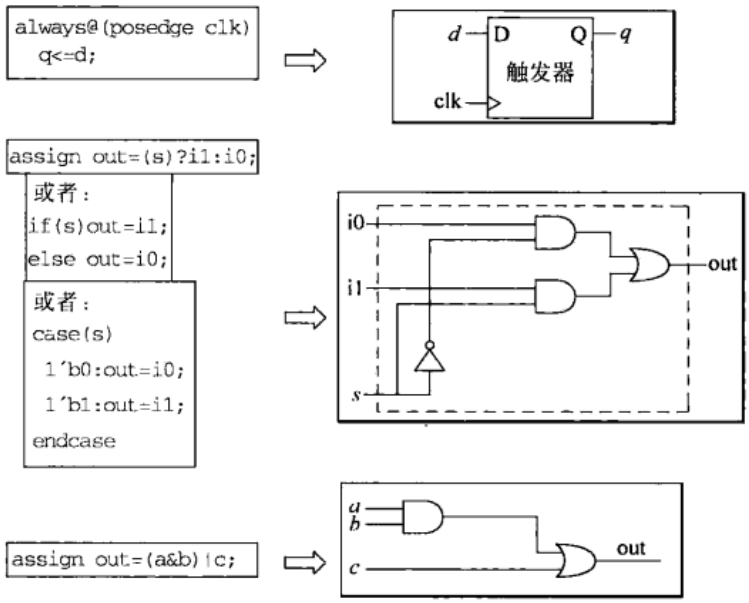


图 3-5 逻辑综合工具会把固定的 Verilog 描述模板转换为相应的电路结构

例如，图 3-6 是美国 IEEE/ACM 组织于 2002 年在 IWLS(international workshop on logic synthesis)国际会议上发布的一个基准测试电路描述。这个电路很简单，名称为“shiftreg_synth”，包含两个输入、一个输出，其中一个是时钟输入端。这个电路 Verilog 代码中的“\”类似于 C 语言中的转义字符，表示从它开始到空格字符为止中间的任何字符串都构成合法标识符。例如“\[11]”表示“[11]”是合法标识符，而不是作为总线或数组中的一位。另外，这个电路共有三个存储单元，初始值为“0”，每次时钟上边沿存储新数据。这些新数据是由一些组合逻辑计算得到。这个 Verilog 代码经过逻辑综合工具进行代码分析后，根据如图 3-5 所示的模板规则，就构建得到如图 3-7 所示的初始电路结构。注意到 Verilog 代码中的各种逻辑运算都直接转换为相应的逻辑门。三个一位存储单元转换为三个触发器 v1、v2、v3。图中标出了代码中给定的输入输出、线网、存储单元名字。因此，这个初始电

路结构和 Verilog 代码之间存在着很清楚的对应关系。但是，并不是所有的 Verilog 代码都可以转化为硬件单元。例如代码中的“initial”语句设定各存储单元的初始值。而在硬件触发器中，这个初始值在硬件上电后可能是随机值。因此像 RTL 代码中“initial”、“time”、“real”等语句都只是用于仿真，而不能构建或综合成硬件单元。对于图 3-6 中的 Verilog 描述，如果要初始化存储单元值，那么可以添加一个“reset”输入端进行复位。另外我们可以注意到图 3-7 中的电路具有图 3-4 所示的典型 RTL 逻辑电路结构。

```
module \shiftreg_synth(v0,clk,\v4.3);
input v0,clk;
output\v4.3;
reg v1,v2,v3;
wire \[11],\[13],\v4.0 ,\v4.1,\[9];
assign
   \[11] = \v4.1,
   \[13] = v0,
   \v4.0 = (~v3&v2)|(v3&~v2),
   \v4.1 = (~v3&v0)|(v3&~v0),
   \v4.3 = v1,
   \[9] = \v4.0;

always @(posedge clk)
begin
  v1 = \[9];
  v2 = \[11];
  v3 = \[13];
end

initial begin
  v1 = 0;
  v2 = 0;
  v3 = 0;
end
endmodule
```

图 3-6　IWLS 2002 测试例子 Verilog 代码[7]

在图 3-3 所示的流程中，RTL 代码经过构建后就得到一个初始的逻辑电路，接下去就对这个电路进行逻辑优化，即在保持电路的输入输出功能不变的前提下，根据布尔代数性质，尽量减少电路内部的逻辑单元数目、输入输出延时、功耗等性能指标。这个步骤一般称做“工艺无关(technology independent)的逻辑电路优化”。在学术界，AIG (and-inverter graph)图是一种非常有效的工艺无关电路的表达方式，它可以表达任意复杂的组合逻辑功能。由于 AIG 表达法的简洁性，它可以很有效地进行各种逻辑变换和优化操作。例如，图 3-8(a)是利用 UC Berkeley 的 ABC 开源工具根据图 3-6 中的 Verilog 描述所生成的 AIG 图。和图 3-7 相比，图 3-8(a)同样包含三个触发器：v1L、v2L 和 v3L，其中触发器的名字后缀“L”在 ABC 中取自 Latch 的首字母，意为时序触发器。但是电路中的组合电路都是用 AIG 的形式，其中节点表示一个 AIG 运算，连线表示各运算的连接关系。实线表示直接相连；虚线表示取反连接，即 AIG 输出经过取反后再连到下级门的输入端。注意到时钟“clk”信号虽然连接到时序存储单元的时钟端，但是在 AIG 中是悬空的。这是因为在芯片设计中，时钟信号直接影响到电路的工作速度。时钟信号在版图中一般和其他逻辑信号是分开处理的，这样可以保证时钟信号能准确地到达各存储单元的时钟输入端，避免产生较大的时钟偏差(clock skew，有时也称为时钟偏移)和时钟延迟(clock latency，有时也称时滞)。对于专用芯片来说，在后端的布图软件中，电路布局后

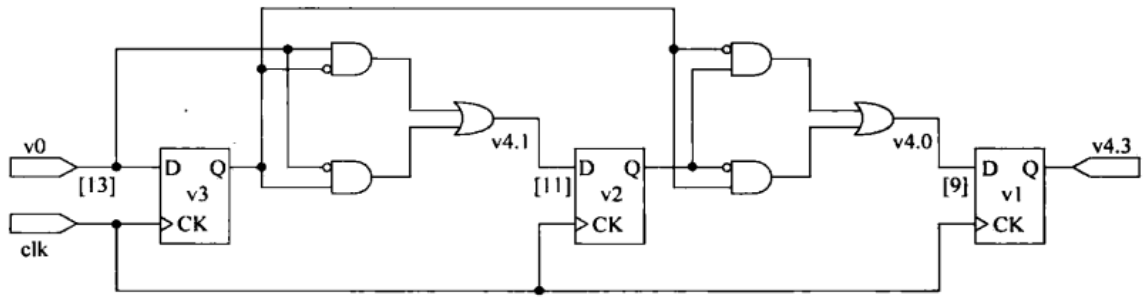
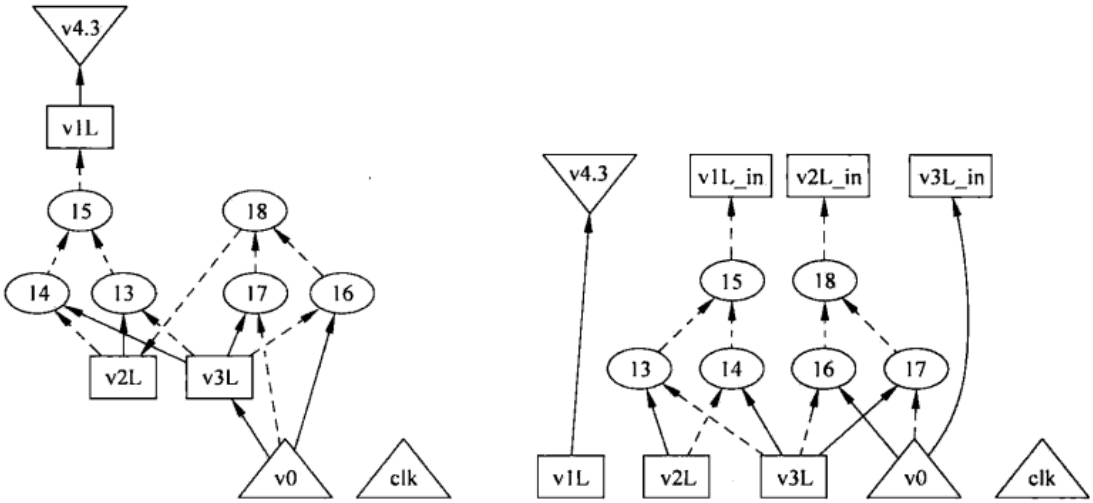


图 3-7　由逻辑综合工具构建的初始电路结构

一般有专门的时钟树综合工具(clock tree synthesis)保证产生最佳的时钟树结构。在FPGA芯片版图设计时,一般也有专门的金属层是给时钟树用的,这样就能保证得到所需要的时钟树优化结构。因此在逻辑优化时,可以暂时不考虑时钟树的互连连接。



(a) 时序电路的AIG表示　　(b) 纯组合电路的AIG表示

图 3-8　ABC 构建的 AIG 时序电路和转换后的纯组合电路

对于电路设计者和逻辑优化来说,触发器或者寄存器的位置和数目在逻辑优化前后一般不会改变。要达到这个目的,它们的输入端一般作为组合电路的输出端,它们的输出端作为下级组合电路的输入端。在优化完成后,再把所有的触发器或者寄存器放回到原来的位置。例如,图 3-8(b)是暂时拿去触发器的 AIG 图,其中 v1L_in、v2L_in 和 v3L_in 是新增的对应于三个触发器输入端的组合输出端;而 v1L、v2L 和 v3L 是新增的对应于三个触发器输出端的组合输入端。这种 AIG 图的优点是后续的优化过程只需要关注图中的纯组合逻辑部分,避免考虑时序单元的存在。在组合电路优化完成后,只要在 v1L_in、v2L_in、v3L_in 和 v1L、v2L、v3L 之间插入三个触发器就可恢复原始时序电路的功能。由于图 3-8 中的电路非常简单,因此工艺无关的门级电路优化并不能减少任何的 AIG 数目或者是输入到输出所经过的级数。因此对这个简单的电路来说,图 3-8(b)也就是优化后的电路结构。

根据图 3-3 的设计流程,优化后的门级电路需要进行工艺映射(technology mapping),就得到相对于某一个工艺库优化的门级电路,如图 3-9 所示。在工艺库中包含了芯片制造商针对某一种半导体工艺所能提供的各种逻辑单元和 IP 核。因此工艺映射的过程就是把工艺无关(technology-independent)逻辑优化后的逻辑电路映射到工艺库中的各种逻辑单元,从而针对具体的工艺库实现了用户电路功能。

对于图 3-8 中的时序电路,ABC 工具利用通用的工艺库进行工艺映射。通用工艺库只包含了基本的逻辑门,例如“与”、“或”、“非”、“异或”等逻辑门。ABC 工具工艺映射后的结果如图 3-10 所示,图中分别给出了 ABC 中“show”命令所显示的结果及其电路图形式。其中图 3-10(a)每个逻辑节点上方的标记表示对应的逻辑单元名字,buf、xor 分别表示缓冲器和异或门。每个节点内部除了节点标号外,还给出了函数信息。例如“01 1”和“10 1”这两行表示 xor 门,即两个输入变量输入相异时输出为 1。这样工艺映射后的电路就能实现

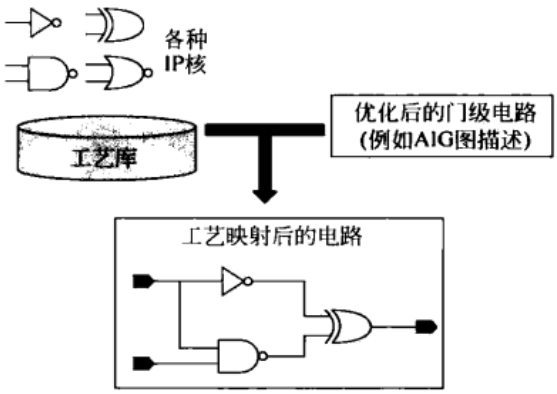
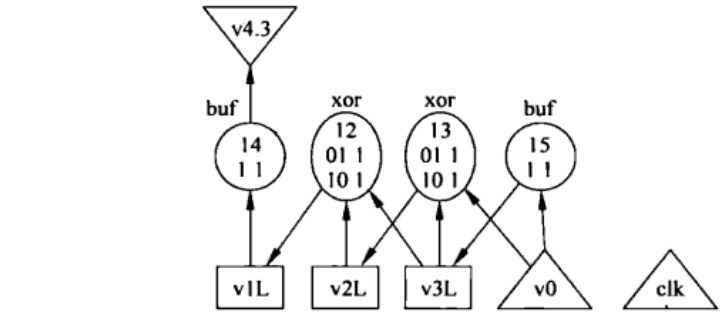
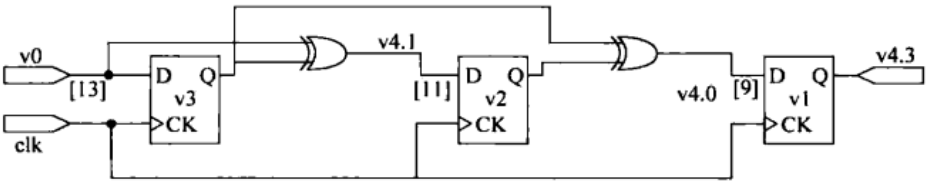


图 3-9　工艺映射基本流程

图 3-6 中 Verilog 代码描述的功能。由于缓冲器对电路的功能没有影响，因此在图 3-10(b)的电路图中并没有画出两个缓冲器。



(a) ABC工具的显示结构



(b) 电路图形式

图 3-10　工艺映射后的电路

如图 3-3 所示，在得到工艺映射后的结果后，前端的综合流程就已结束。接下去就开始后端的专用芯片布图流程[5]。值得一提的是，如果我们比较图 3-1 和图 3-3 两个流程，可以发现如果能够连接软件编译器输出的 RTL 描述和 EDA 前端综合工具输入的硬件 RTL 描述，那么就可以直接从高级语言，例如 C/C++、Java、Fortran，直接生成高级语言描述的芯片版图，这样可以极大地提高芯片设计的效率——用户不需要学习硬件描述语言就可以完成芯片的自动化设计。不过这方面的工作一般称做高层次综合，目前还只是处于研究阶段，并没有非常成熟的自动化产品。有兴趣的读者可以参见文献[8]及后续大量关于高层次综合的研究论文。

3.2 基于 FPGA 的数字电路设计流程

FPGA 作为通用的现场可编程逻辑阵列器件，能够在芯片制造完成后，通过下载不同的二进制文件——位流文件实现不同的芯片功能。基于 FPGA 的验证和测试流程首先读取用户的 RTL 描述，然后进行代码分析，得到用户电路的编译器内部表达方式。接下去编译器就进行工艺无关和工艺相关(technology dependent)的逻辑优化和工艺映射，输出优化后的门级网表。在后续的 FPGA 验证中，优化后的逻辑电路还要映射到 FPGA 中的可编程逻辑单元，再经过布局布线后产生位流文件。用户把位流文件下载到 FPGA 芯片后就能进行实际的电路测试。同时，FPGA 编译器要对布局布线后的电路进行时序和功耗分析，输出报告文件。这种可编程方式与通用处理器的工作原理和编译流程非常相似，两者都需要编译器来产生可下载的二进制文件。它们的不同之处是：前者为硬件可编程，其可编程文件一般下载到芯片的内部存储单元；而后者一般称为软件可编程。因为通用处理器内部的寄存器和高速缓冲器一般不能直接被用户所用，所以软件可编程文件一般在程序运行时被调到主内存。通用处理器的可编程存储资源管理要比 FPGA 复杂很多，除了内存外一般都包含多级高速缓冲器和寄存器组。并且应用程序中的指令必须经过处理器内部的指令译码器才能配置 ALU 单元完成所需要的操作。FPGA 内部的存储单元分为两种。一种用于存储可编程信息。这部分存储单元在芯片工作时一般是不可改变的，为只读存储单元。它类似于通用处理器中的只读可执行文件。对于芯片运行时动态改变本身配置的 FPGA 系统一般称为可重构计算系统，此部分内容参见第 6 章。另一种是用于存放用户动态数据的存储单元。早期 FPGA 器件用户只能利用 DFF 或 LUT 存储用户数据。但是国际上近期商用系统级 FPGA 器件所包含的存储单元越来越丰富，规模也越来越大，可以直接存放大规模的用户数据。另外系统级 FPGA 一般还具有嵌入式通用处理器核，直接可以运行一般的软件程序。因此系统级 FPGA 设计对用户的软硬件协同设计提出了更高的要求，它所发挥的功能也越来越强大。

下面基于 3.1 节介绍通用处理器的编译过程和基于 EDA 工具的数字集成电路设计流程来理解 FPGA 位流文件生成的基本原理。

如图 3-11 所示，FPGA 位流生成流程和图 3-3 的专用芯片设计流程很类似，也分为前端综合和后端布图与位流生成两大部分。两者的主要差别如下。

(1) 在工艺映射阶段，专用芯片设计流程是把优化后的门级网表映射到给定的工艺库中的各逻辑单元或 IP 核。由于 LUT 是 FPGA 可编程逻辑的核心，FPGA 的工艺映射是把用户电路映射到可编程逻辑单元 LUT 及与之相连的其他逻辑门，例如 MUX、XOR 等。根据第 2 章的讨论，一个 n 输入的 LUT 能够实现 2^{2^n} 个逻辑函数，在工艺映射时不可能把这些 2^{2^n} 个所能实现的逻辑函数功能全部列出然后再进行映射。因此专用芯片设计中具有固定功能的逻辑单元映射算法和基于 LUT 的映射算法两者之间具有很大的差别。这种差别主要是由于 LUT 的通用灵活性所引入的。

(2) 对于专用芯片来说，后端布图 EDA 工具的主要目的是在满足用户对芯片性能的约束前提下，尽量减少芯片的版图面积，从而降低芯片的制造成本和可靠性、功耗等性能。由于 FPGA 是事先制造完成的芯片，即用户在设计完成后不需要重新流片，因此后端布图软

件的主要目的是充分利用 FPGA 芯片已经提供的各种逻辑资源、互连资源和其他专用 IP 核，实现用户对芯片性能的各种约束，例如速度、功耗等。由此可见，FPGA 的布局布线算法和专用芯片的布局布线算法具有本质的差别。

（3）专用芯片设计完成后输出的芯片版图文件，供芯片制造商完成芯片的制造。但是对 FPGA 来说，设计完成后所生成的是位流配置文件。用户只需要把位流文件下载到 FPGA 内部，就能实现电路功能。从这个角度来说，FPGA 的位流文件等价于高级软件语言编译器所产生的可执行程序。另一方面，由于 FPGA 可编程性和灵活性的代价，同一个 RTL 电路描述在相同工艺条件下，利用 FPGA 所实现的性能往往比专用芯片差数倍甚至数十倍[9]。但是随着半导体工艺技术的不断更新换代，芯片的制造成本呈指数上升。高额的开发成本要求芯片有大量的市场和足够的利润。因此诸如 CPU 和 FPGA 等通用型器件所采用的半导体工艺比专用芯片一般要领先 1～2 代。在深亚微米或纳米级半导体工艺时代，利用最新半导体工艺的 FPGA 的灵活性和低 NRE 成本具有明显的优势，特别是对于那些中小批量规模的应用芯片。最近 FPGA 开始进入终端产品领域，而不局限于数字电路的原型设计验证。

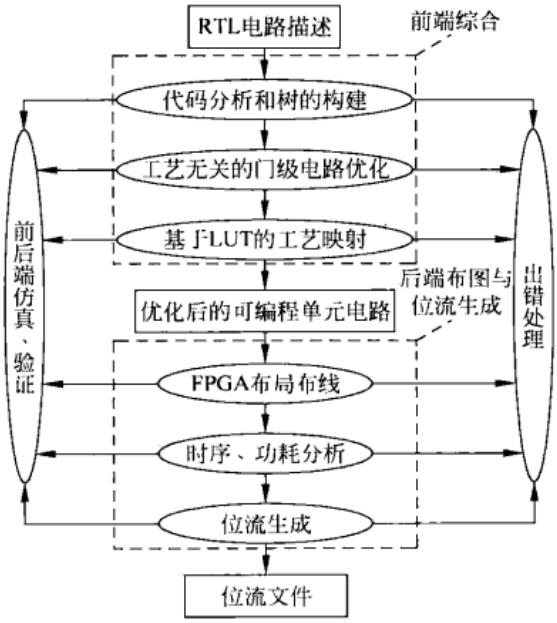


图 3-11　基于 FPGA 的数字电路设计流程

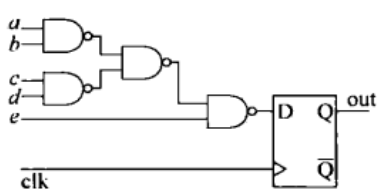
从整体设计流程来看，图 3-11 基于 FPGA 的数字电路设计和图 3-3 专用芯片的设计，以及图 3-1 高级语言编译流程都有类似之处。从通用性的角度来看，如果能够把高级语言编译流程中产生的 RTL 描述转换成 FPGA 数字电路设计的输入格式，那么就可以实现从高级语言到硬件实现的整体流程。考虑到熟悉高级语言的用户远远要超过熟悉硬件描述语言的用户，并且硬件电路实现达到的性能要远远超过在通用处理器上实现同一个数字计算任务所能达到的性能，如果能在软件流程上克服连接两种 RTL 描述之间的障碍，那么就能进一步提高设计效率和产品性能，充分利用 FPGA 提供的可编程硬件资源和嵌入式通用处理器核，提升系统级 FPGA 的认知度。目前在高层次综合领域，即能够直接把高级语言描

述的程序直接转化为在 FPGA 上能够直接运行的硬件语言的描述，在国际上还没有非常成熟的产品。

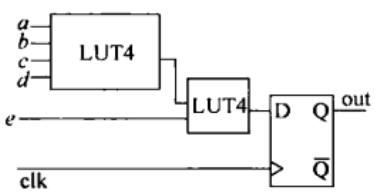
3.3　基于 LUT 的 FPGA 工艺映射

根据图 3-11 的流程，基于 FPGA 的工艺映射和专用芯片的工艺映射过程主要的区别在于所面向的映射对象：前者是把逻辑优化后的门级电路映射到 FPGA 的基本可编程逻辑单元，它和不同 FPGA 器件的结构相关；后者是把门级电路映射到指定半导体工艺库中的各种逻辑单元或者专用 IP 核，它和用户所选择的后端半导体工艺相关。基于第 2 章的分析与比较，LUT 是主流 FPGA 器件的基本可编程逻辑结构，并且传统上四输入 LUT 在面积利用率和速度方面的综合性能最佳[10]。因此如何把一个门级组合电路在满足最少级数的 LUT 的前提下用最少的 LUT 实现就是 FPGA 工艺映射的关键问题。其中最少级数的 LUT 表示电路从组合输入到输出的最大延时最小；最少 LUT 表示电路尽可能节约对硬件资源的使用，从而让一个 FPGA 器件能够容纳更大规模的用户电路。因此 FPGA 工艺映射问题就是在满足最小延时的约束下优化所占用的 LUT 个数，避免 LUT 资源不必要的浪费。对于商用器件来说，FPGA 工艺映射问题就是在满足最小延时的约束下优化所占用的可编程逻辑单元个数，并且尽可能提高用户电路的工作速度。

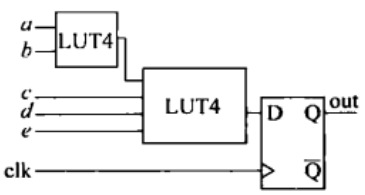
图 3-12(a)是一个简单的逻辑电路经过工艺无关的逻辑优化后的结果，其中包含 1 个触发器和 4 个与非门。如果我们允许把 AIG 看成是一种可编程门，那么与非门就是 AIG 门的特例，即 AIG 门的两个输入不取反，输出取反。这个电路共有 6 个输入端和 1 个输出端。由于时钟端 clk 一般都是利用专用的时钟树资源进行单独布线，并且触发器在 FPGA 器件中已经独立存在，因此工艺映射的主要任务就是把由 4 个与非门组成的组合逻辑用 LUT 来实现。因为这个电路的组合逻辑部分共有 5 个输入端：a、b、c、d、e，一个输出端连到触发器的 D 输入端，所以它至少需要两个 LUT4 单元才能完成映射。这是因为一个 LUT4 可编程单元只能实现任意的四输入组合函数。图 3-12(b)给出了一种映射后的电路结构。它把 a、b、c 和 d 四个输入映射到一个 LUT4，然后再把这个 LUT4 的输出和输入端 e 一起映射到



(a) 逻辑优化后基于AIG的门级电路



(b) 一种映射结果



(c) 另一种映射结果

图 3-12　基于 LUT 工艺映射示例

另一个 LUT4。这是一种可行的映射结果。但是考虑到电路速度的约束需求，这种映射后的电路不一定是最佳结构。假设从前级电路输入到 a、b、c、d 和 e 的延时是不同的，例如 c、d 和 e 的延时要大于 a 和 b 的延时，那么从关键路径延时考虑，图 3-12 (c)就是一种更好的结构。从以下的计算示例中可以作简单的比较。

假设在图 3-12 中，从前级电路输入到 a、b、c、d 和 e 的延时分别是 1ns、1ns、2ns、2ns 和 2ns，并且一个 LUT4 本身的延时是 3ns，从第一个 LUT4 输出到第二个 LUT4 输入的互连延时为 0.4ns，从 LUT 输出到触发器 D 端的局部互连延时为 0.1ns。那么各个路径到触发器 D 端的组合延时如下所示。

对于图 3-12(b)的电路，可以计算如下。

a、b 端到 D 端延时：1+3+0.4+3+0.1=7.5(ns)

c、d 端到 D 端延时：2+3+0.4+3+0.1=8.5(ns)

e 端到 D 端延时：2+3+0.1=5.1(ns)

因此 D 端的最大延时为 8.5ns。

对于图 3-12(c)的电路，可以计算如下。

a、b 端到 D 端延时：1+3+0.4+3+0.1=7.5(ns)

c、d 端到 D 端延时：2+3+0.1=5.1(ns)

e 端到 D 端延时：2+3+0.1=5.1(ns)

因此 D 端的最大延时为 7.5ns，它比图 3-12(b)的电路减少了 1ns 的延时。可见基于 LUT 的工艺映射需要综合考虑 LUT 数目和延时等因素，从而优化电路的综合性能。一般来说，FPGA 的工艺映射算法主要目标是在满足用户给定的延时约束或者是在延时最优化的情形下，尽量减少所用的 LUT 数目。为了达到这样的目标，在工艺映射第一步优化的情况下得到了最短的 LUT 级数，然后把这个级数作为时序约束运行再综合(resynthesis)算法使得在最短 LUT 级数的约束下进一步减少 LUT 的数目。以下简单介绍 FPGA 工艺映射的常用算法。

3.3.1　枚举算法

枚举法是工艺映射的基本方法，其最主要的思想就是对工艺无关逻辑优化后的 AIG 电路通过枚举所有的割集(cut 集)的方法，确定输入电路的一种划分方式。一个割集就是分割(cut)的集合，“分割”也简称为割。定义割集的标准就是割集中的每一个分割的输入都要小于或等于 LUT 的输入数 K，并且这个割集能够覆盖电路中所有的逻辑。我们将覆盖这个电路图中的每一个 K 输入分割称为 K-cut。这样一个 K-cut 可以由 FPGA 硬件资源中的 K 输入 LUT 来实现。一个电路经过划分和优化后得到的 K-cut 的数量就是所需要的 LUT 数目。枚举法的实现比较简单，理论上只要穷举所有的分割就可以找到 LUT 级联数目最少的工艺映射结果，但是枚举分割的数量会随着电路规模的扩大而急剧增加。

在进行工艺映射之前，首先要将输入电路转化为有向无环图(directed acyclic graph，DAG)的描述形式。将电路的逻辑门抽象成图中的一个节点，而将门电路之间的连线关系转化成有向边，这些有向边是从逻辑门的输出指向下一级的输入。图 3-12(a)中的电路转换后的 DAG 图如图 3-13 所示，其中圆圈里的数字表示逻辑节点的标号。

基于割集的枚举算法进行工艺映射的过程可以分为以下几个步骤。

1. 枚举每个逻辑节点的割集

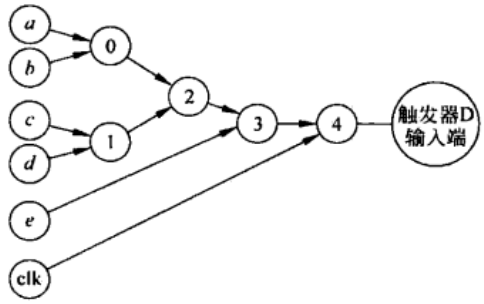


图 3-13 电路转换后的 DAG 图

遍历整个电路的 DAG 图，按照从输入到输出的顺序枚举所有逻辑节点的割集。若当前节点为输入或输出节点，则只需要建立一个 cut，所含节点就是它们本身。若当前节点为内部逻辑门对应的节点，建立 cut 的顺序为：首先建立一个单独的 cut，该 cut 只包含当前节点本身。这种 cut 的级数(level)和面积(area)成本都定为某个预先设置的极大值，确保其被选取的优先级最低；其次根据当前节点的前级输入节点上的所有 cut 判定是否并入本 cut 从而生成新的 cut。判定的规则是新 cut 所对应的输入个数(注意并不是节点个数)不能超过 K，即 LUT 的输入个数。由于 K 输入的 LUT 能够实现任意的 K 输入函数，因此只要输入数目不超过 K，不管函数逻辑多么复杂，它都能用一个 LUT 来实现。另外由于产生割集的顺序是从输入到输出，因此遍历一次后总能枚举所有有效的割集。枚举阶段算法的伪代码如图 3-14 所示。

```
procedure cutEnumeration (G);
1  begin
2    vNodes←get_node_by_DFS(G)
3    for each node u∈ vNodes do
4      enumNode(u);
5  end;
procedure enumNode (u);
1  begin
2    new_cut←create_a_cut(NIL,NIL,u)
3    push new_cut to vector allcuts[u]
4    for each cut c1∈ allcuts[left_child[u]] do
5      for each cut c2∈ allcuts[right_child[u]] do
6      begin
7        new_cut←create_a_cut(c1,c2,u)
8        if num_of_input[new_cut]≤K
9          push new_cut to vector allcuts[u]
10     end
11 end;
```

图 3-14 根据 DAG 图枚举所有 cut 的伪代码

2. 找到各节点的最优割

得到所有节点对应的割集之后，为了保证时序性能，需要从各节点的割集中选取延时最优的割集。在工艺映射阶段，时序性能最直接的表示就是网表的级数，因此时序最优意味着用最少级数的 LUT 来实现电路。采用深度优先的次序遍历所有节点，对每个节点的割集内所有的 cut 进行贪婪分析，从而确定每个节点对应的最优 cut，根据该 cut 的面积和级数进行选取，而面积和级数的计算都是根据该 cut 对应的输入节点上的所有最优 cut 确定的。

以级数的计算为例，当前 cut 的级数值为其所有输入节点对应最优 cut 的级数值加 1。伪代码如图 3-15 所示。

```
procedure cutSelection (G);
1 begin
2   vNodes←get_node_by_DFS(G)
3   for each node u∈vNodes do
        selcut(u);
end;
procedure selcut(u);
begin
   bestcut = allcuts[u][0]
   for each cut c∈allcuts[u] do
   begin
      c.level = MAX(bestcut[c.inputs].level) + 1
      c.area = TOTAL(bestcut[c.inputs].area) + ONE_LUT_AREA
      if c.level < bestcut.level
          bestcut = c;
      else if (c.level = bestcut.level) and (c.area < bestcut. area)
          bestcut = c;
   end
   bestcut[u] = bestcut
end;
```

图 3-15　从 cut 集合中选择各节点的最优割的伪代码

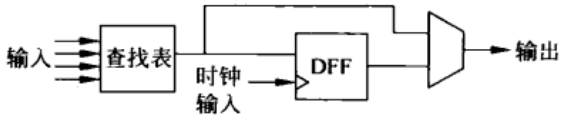
3. 确定整个电路的最优割集

在确定了所有节点所对应的最佳 cut 后，就需要从各个节点的最优 cut 中选取部分作为整个电路的最终 LUT 所对应的 cut 集合。选取的方法仍然是基于贪婪算法，即从输出的位置开始向输入的方向选取。首先选取输出节点所对应的最优 cut，然后再选取该 cut 所对应的前级输入节点的最优 cut，直至覆盖整个电路。至此，已经将整个门级电路全部映射到 LUT，其中的任意一个 LUT 都分别对应于一个 cut。

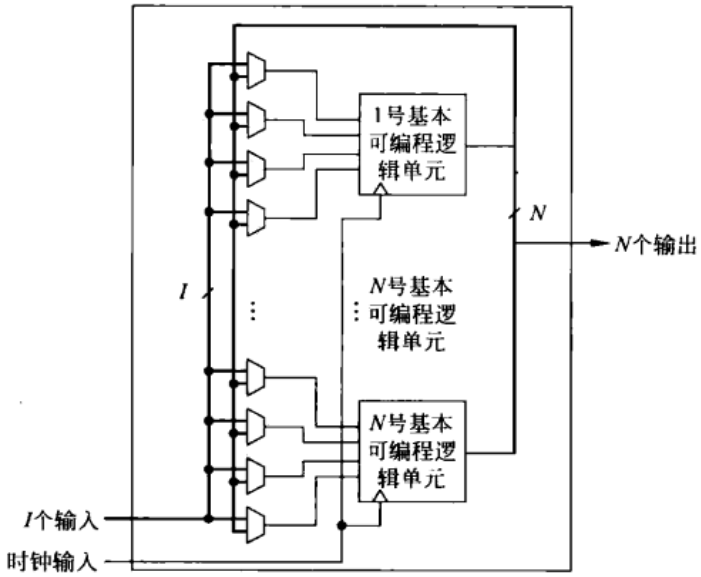
3.3.2　逻辑单元块打包

逻辑单元块打包问题，即把工艺映射产生的 LUT 和触发器或者寄存器打包成一个可编程逻辑单元，或者把数个可编程逻辑单元根据 FPGA 的硬件结构打包成一个可编程逻辑簇的过程。学术界采用的是一种基于簇结构的统一逻辑单元模型。这种模型是一个双层次的结构：第一个层次是基本可编程逻辑单元(basic programmable logic element，BLE)，它包括一个 LUT 和一个触发器。而整个簇是由多个基本逻辑单元构成，它就构成了第二个层次，如图 3-16 所示。目前被广泛应用的打包工具是由 Toronto 大学开发的 T-VPack，通过选取关键路径上的基本逻辑单元作为种子进行打包，然后再利用时序驱动成本函数选取与该种子单元直接连接的 BLE 继续打包，从而获得良好的时序性能。具体打包算法可以参见文献[11]。

现代商用 FPGA 的逻辑单元结构远比这种基于简单簇结构的模型要复杂。图 3-17 所示为一种典型的可编程逻辑单元块的结构图。从图中可以看出，除了标准的查找表和触发



(a) 基本可编程逻辑单元结构



(b) 逻辑簇结构

图 3-16 逻辑簇的双层次结构[11]

器外，这种结构还增加了其他更为灵活的逻辑功能。例如对于一位全加器来说，它有三个输入，两个输出。由于一个四输入查找表只有一个输出，因此如果不添加辅助逻辑的话，一位全加器就需要两个四输入查找表来实现。此时每个四输入查找表都是作为三输入函数使用，从而降低了硬件资源的利用率。因此为了提高多位加法器的性能，商用 FPGA 的可编程逻辑单元一般都会增加进位链结构，利用额外的进位逻辑生成进位输出。同时可以利用逻辑簇内部的快速互连，产生快速进位输出，如图 3-17 所示。另外注意到图中还有两个控制 MUX 的码点，它们用于控制两个查找表的组合输出方式。即只要控制这两个码点，就可以在两个组合输出端得到内部任意一个查找表的输出。但是如果把这两个码点中的其中一个或者两个都改为外部输入，那么这两个外部输入和两个四输入查找表就可以构成更大尺寸的查找表。例如如果把“码点 1”改为外部输入，那么图中的两个四输入查找表就可以合并成为一个五输入查找表。第五个输入就是对应于“码点 1”的新输入端。再且，如果同时把“码点 2”也改为输入端，那么第六个输入端“f”就可以从逻辑簇内部的其余五输入查找表通过内部反馈输入，这样四个四输入查找表就构成了一个六输入查找表，从而可以在一个逻辑簇内部实现快速逻辑。这比经过逻辑簇外部互连实现的六输入函数要快很多。不过这样做的代价是增加了逻辑簇内部的局部互连资源需求。同时如果六输入查找表的第五和第六个输入不在本逻辑簇内部产生，那么还要利用逻辑簇外部的互连资源把六输入查找表的第五和第六个输入引入到逻辑簇内部，这样增加了互连资源的压力。因此如何识别出电路网表中的各种逻辑功能，并且将其打包到相应的逻辑单元中，从而保证电路性能和所使用

FPGA 硬件资源的优化结果，这些对打包算法提出了新的挑战。

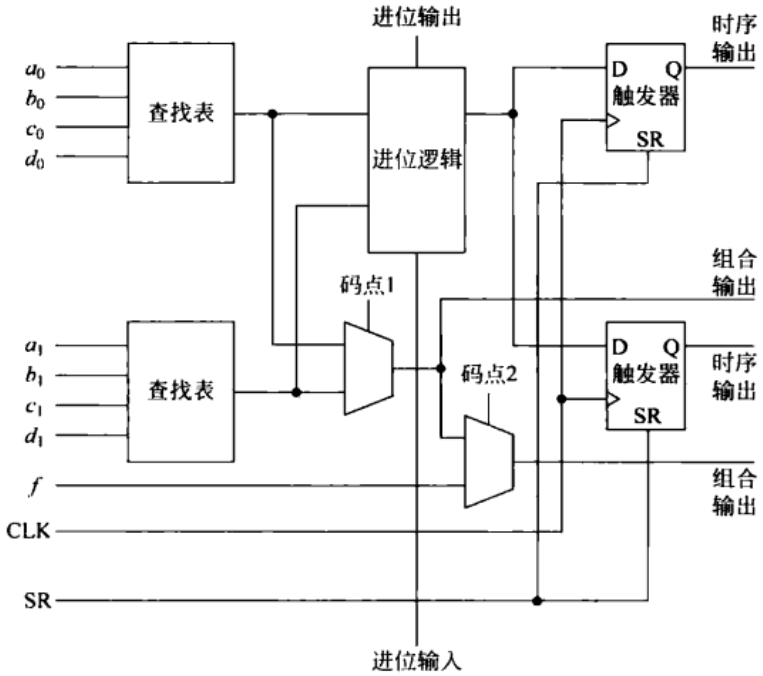


图 3-17　典型的复杂可编程逻辑单元结构

3.3.3　逻辑再综合

对于基于 AIG 图描述的数字电路来说，虽然它经过了工艺无关的逻辑优化，但是在实践上由于算法复杂度和运算时间的限制并不能保证得到最优的电路结构。为了突破电路结构的限制，在基于 LUT 的工艺映射算法中，往往还需要采用基于布尔函数分解(functional decomposition)或者其他更为复杂的优化方法进行电路重综合(resynthesis)，以进一步优化电路的性能或者在不增加电路级数的前提下减少查找表的使用数目。一个 n 变量函数 f 的香农展开式就是最基本的函数分解形式：

$$f(x_0x_1\cdots x_{n-1}) = \bar{x}_i f(x_0x_1\cdots 0\cdots x_{n-1}) + x_i f(x_0x_1\cdots 1\cdots x_{n-1}) \tag{3-1}$$

或者缩写为

$$f(x_0x_1\cdots x_{n-1}) = \bar{x}_i f_{\bar{x}_i} + x_i f_{x_i} \tag{3-2}$$

其中 $f_{\bar{x}_i}$ 和 f_{x_i} 分别表示当 x_i 为 0 和 1 时 f 的两个余函数(cofactor)。下面以一个常用的四选一多路选择器为例说明再综合所取得的效果。

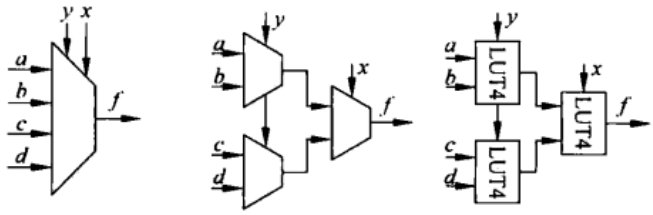
图 3-18(a)是四选一数据选择器的输入输出接口，它共有 6 个输入端，1 个输出端。根据它的逻辑意义来分析，4∶1-MUX 可以很直观地由 3 个 2∶1-MUX 来实现，如图 3-18(b)所示。根据这样的函数分解方法进行映射就需要 3 个 LUT4，其中每个 LUT4 均用于实现三输入变量的 2∶1-MUX 函数，如图 3-18(b)所示。这种工艺映射结果的资源利用率不高，即每个 LUT 均有一个输入是浪费的，只作为 LUT3 使用。但是如果我们引入如图 3-18(c)所示的四变量函数对 4∶1-MUX 进行香农分解，那么函数 f 就可以分解为

$$f = \bar{g}f_{g=0} + gf_{g=1} = \bar{g}f_{\bar{g}} + gf_g \tag{3-3}$$

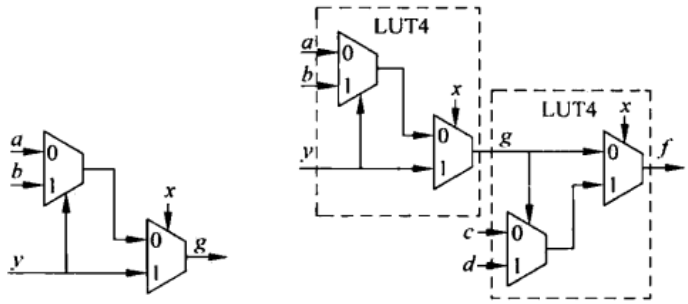
或

$$f = (\bar{g} + f_{g=1})(g + f_{g=0}) = (\bar{g} + f_g)(g + f_g) \tag{3-4}$$

其中 g 是与变量(a,b,x,y)相关的四变量函数，如图 3-18(c)所示。$f_{g=0}$或 f_g，$f_{g=1}$或 f_g 分别表示当 g 为 0 和 g 为 1 所对应的各组(a,b,x,y)取值时 f 的余函数。此时 4∶1-MUX 就可以分解为两个输入函数的级联。这样就可以用两个 LUT4 来实现，如图 3-18(d)所示。因此如何根据图 3-18(b)中的前期工艺映射结果，识别函数的冗余性和相关性，从而找到合适的再综合方法，提高 LUT 资源的利用率，就成为再综合算法的主要目的。



(a) 四选一MUX的接口　(b) 利用二选一MUX分解及其相应的LUT电路



(c) 引入新函数来进行分解　(d) 四选一MUX映射成两个LUT4

图 3-18　4∶1-MUX 的工艺映射结果

3.4　时序驱动的布局布线和物理综合时序优化技术

进行工艺映射后的逻辑电路只包含了 FPGA 硬件所提供的可编程单元，即工艺映射算法已经把用户电路映射到 FPGA 的可编程单元了。这样下一步的目标就是把这些硬件可编程单元放到 FPGA 器件内部，用互连资源进行连接，并且能够优化电路的性能。这一步骤就是布局布线。

3.4.1　时序驱动布局与布线

布局算法确定了实现电路功能需要的各逻辑单元在 FPGA 中的位置；而布线算法负责选择合适的可编程开关以连接电路需要的所有逻辑单元。时序驱动布局与布线器则以最大限度提高电路速度作为优化目标。以下就按照布局布线过程的主要组成部分进行讨论，更详细的描述可参见文献[11]。

1. 时序引擎

时序引擎通过时序图来估算布局与布线过程中源端到漏端连接的延时裕量(slack)以

及整个电路的关键路径位置和最大延时。时序图是描述电路结构的有向图。时序图中每个节点表示诸如寄存器和查找表之类的基本单元电路的输入输出引脚；每条有向边表示诸如查找表的组合逻辑单元的输入和输出之间的连接，或者是电路网表所规定的各个可编程单元之间的连接。每条边上都标注通过单元电路或者布线资源所需要的延时。

时序图为图论中的事件节点图，为了确定最小时钟周期，可以对带 n 个节点的时序图进行广度优先遍历。从没有输入边的节点，即主输入节点开始遍历，并且把它们的信号到达时间 T_{arrival} 标注为 0。对于未标注的后级节点，由于它的输入边所对应的连接均已标注，因此它的到达时间可用下式表示：

$$T_{\text{arrival}}(i) = \text{Max}_{\forall j \in \text{fanin}(i)}\{T_{\text{arrival}}(j) + \text{delay}(j,i)\} \tag{3-5}$$

其中节点 i 是当前被标注的节点，delay(j, i)是节点 j 到 i 连接边的延时值，连接边的方向是从 j 指向 i。根据式(3-5)就可以把整个图中的所有节点均进行标注。最大到达时间所在的节点，往往会是主输出或者寄存器的输入端。这个节点所对应的时间就定义了整个电路中的最大延时 D_{max}。这个最大到达时间上的各个延时组成部分就构成了电路的关键路径，用于确定电路的最高工作时钟频率。

虽然由上述步骤就可以确定电路的速度，但是在时序驱动布局布线算法中，还需要衡量各源端到漏端连线影响电路延时的关键度。某一连接的延时裕量可定义为在不增加电路最小时钟周期的前提下允许被添加到这条连接上的延时量。为了计算延时裕量，就要对时序图再进行广度优先遍历。这时把没有输出边的节点的需求时间 T_{required} 设置为 D_{max}，并且对图进行从输出到输入方向的广度优先遍历以反标剩余的节点。带扇出的节点 i 的需求时间为

$$T_{\text{required}}(i) = \text{Min}_{\forall j \in \text{fanout}(i)}\{T_{\text{required}}(j) - \text{delay}(i,j)\} \tag{3-6}$$

于是从节点 i 到 j 连接的延时裕量为

$$\text{slack}(i,j) = T_{\text{required}}(j) - T_{\text{arrival}}(i) - \text{delay}(i,j) \tag{3-7}$$

延时裕量为 0 的连接通路就组成了电路的关键路径——增加这条连接的延时就会相应地导致整体电路延时的增加；相反，延时裕量较大的连线就可以用较慢的路径布线，这并不影响整体电路的延时。至此时序引擎就计算了整个电路的最大延时和各条路径的延时裕量。

2. 时序驱动布局器

传统的布局器主要基于模拟退火算法，算法的关键在于成本函数的计算与评估。成本函数用于评估逻辑单元布局的质量。图 3-19 是基于模拟退火算法布局器的伪代码。

如以上伪代码所示，布局开始时先把逻辑单元随机地分配到 FPGA 以得到一个初始布局。然后随机地选取一个逻辑单元的新位置，同时计算由位置交换所造成的成本函数的差值。若成本函数减少则接受本次交换；然而即使成本函数值增大，位置交换使布局变差，但是仍然存在交换被接受的可能。接受的概率由 $e^{-\Delta C/T}$ 给出，其中 ΔC 就是交换造成的成本函数的变化，T 是温度参数，用于控制接受导致布局变差的位置交换的概率。开始时由于退火温度非常高致使几乎所有的交换都被接受；随着退火过程的进行，退火温度逐渐降低，布局质量逐渐提高，使得接受导致布局变差的交换的概率也逐渐降低。模拟退火过程中对于温度下降的速度、终止退火的退出标准、在每个温度下试图移动的次数和产生可能移动的方法是由退火表决定的。退火表的质量将影响布局的收敛速度以及最终布局结果的质量。

```
S = RandomPlacement ();
T = InitialTemperature ();
Rlimit = InitialRlimit ();

while (ExitCriterion () == False) {          /* "外循环" */
    while (InnerLoopCriterion () == False) { /* "内循环" */
    Snew = GenerateViaMove (S, Rlimit);
    ΔC = ComputeDeltaCost(Snew, S)
    r = random (0,1);
    if (r < e ^(ΔC/T)) {
        S = Snew;
    }
} /* "内循环"结束 */
T = UpdateTemp ();
Rlimit = UpdateRlimit ();
}  /* "外循环"结束 */
```

图 3-19 基于模拟退火算法布局器的伪代码

对于时序驱动布局器，图 3-19 对成本函数差值 ΔC 的计算由下式给出：

$$\Delta C = (1-\beta)^{*}\Delta bb\_cost^{*}(1/\text{pre}\_bb\_cost)+\beta^{*}\Delta t\_cost^{*}(1/\text{pre}\_t\_cost) \tag{3-8}$$

以下分别解释这个公式的各个分量。

(1) β 表示线长驱动与时序驱动二者比例的折中因子。当 $\beta=0$ 时，成本函数的优化目标仅仅是最大限度地减少所需的布线资源；当 $\beta=1$ 时，成本函数的优化目标仅仅是最大限度地提高电路速度。为使最终布局结果达到布线资源数量和时序性能上的平衡点，β 一般取 0.5。

(2) Δbb_cost 表示交换两个可编程逻辑单元所导致的电路线网边界框成本差值；pre_bb_cost 表示交换前电路所具有的线网边界框成本。其中电路所具有的线网边界框成本 bb_cost 由下式给出：

$$bb\_cost = \sum_{i=1}^{N} q(i)\left[\frac{bb_x(i)}{C_{\text{av},x}(i)^{\beta}}+\frac{bb_y(i)}{C_{\text{av},y}(i)^{\beta}}\right] \tag{3-9}$$

上式对电路中的 N 条线网求和。对于每条线网 i，$bb_x(i)$ 和 $bb_y(i)$ 分别表示其边界框的水平和垂直跨度。$q(i)$ 是补偿因子。$C_{\text{av},x}(i)$ 和 $C_{\text{av},y}(i)$ 分别是线网 i 的边界框在 x 和 y 两个方向的平均通道宽度，即互连轨道数量。

(3) Δt_cost 表示交换两个可编程逻辑单元所导致的电路延时成本差值；pre_t_cost 表示交换前电路所具有的延时成本的倒数。其中电路所具有的延时成本 t_cost 由下式给出：

$$t\_cost = \sum_{m=1}^{N_{\text{nets}}}\sum_{n=1}^{N_{\text{sinks}}}\text{Crit}(i,j)\text{delay}(i,j) \tag{3-10}$$

上式中内层求和表示对线网中所有漏端求和；外层求和表示对电路中所有线网求和。delay(i, j)表示线网中源端 i 到漏端 j 的连接延时，该延时值通过对预先建模的延时库查得。Crit(i, j)表示线网中源端 i 到漏端 j 的连接关键度，由下式给出：

$$\text{Crit}(i,j) = 1-\frac{\text{slack}(i,j)}{D_{\max}} \tag{3-11}$$

式中，slack(i, j)以及 $D_{\max}$ 的值由时序引擎估算得出，见式(3-7)及相关的时序引擎的描述。

3. 时序驱动布线器

布线算法是基于布线资源图进行的。图 3-20 是一个简单的二输入二输出可编程逻辑单元和四条互连线的布线资源图示例。其中可编程逻辑单元的输入输出、互连线都用布线资源图的节点表示，布线资源图的有向边表示可编程连接关系。考虑到输入端和输出端的逻辑等价性问题，在布线资源图中一般都会添加源端和漏端。例如在逻辑功能上 LUT 的各个输入可以进行互换，只要更新 SRAM 的配置值即可。因此布线器可以利用这种逻辑等价性进行交换，从而可以减少关键路径的延时。

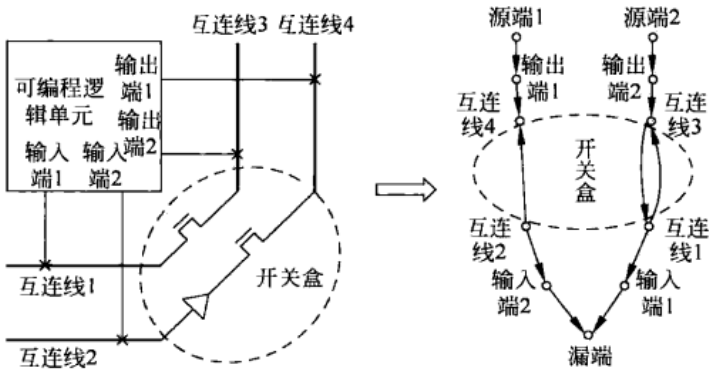


图 3-20　布线资源图示例[11]

在建立好布线资源图之后，运用迷宫算法连接线网中的源端到每一个漏端。迷宫布线算法的本质是运行 Dijkstra 算法，在布线资源图中找到一根线网源端和漏端之间的最短路径。Dijkstra 算法是典型的最短路径算法，用于计算一个节点到其他所有节点的最短路径。其主要特点是以源端为中心向外层扩展，直到扩展到线网的各个漏端为止。最短路径一般是通过执行多次布线迭代来找到的。在迭代过程中，一些线网被拆线重布到不同的路径上，以解决布线资源的竞争或者改进电路的速度。时序驱动布线算法可以采用较为简单的 Elmore 延时模型。它的计算量不大，但是时序效果要比线性延时模型好很多。在 Elmore 延时模型中，线网 i 的第 j 个漏端使用节点 n 的布线成本为[11]

$$\mathrm{Cost}(n) = \mathrm{Crit}(i,j) \cdot \mathrm{delay}_{\mathrm{Elmore}}(n,\mathrm{topology}) + [1 - \mathrm{Crit}(i,j)] \cdot b(n) \cdot h(n) \cdot p(n) \tag{3-12}$$

上式中的各分量意义如下。

(1) 关键度 $\mathrm{Crit}(i,j)$ 由下式给出：

$$\mathrm{Crit}(i,j) = \max\left(\left[\mathrm{MaxCrit} - \frac{\mathrm{slack}(i,j)}{D_{\max}}\right]^{\eta}, 0\right) \tag{3-13}$$

式中 $D_{\max}$ 和 $\mathrm{slack}(i,j)$ 是由时序引擎估算得出。η 和 MaxCrit 是成本函数式中控制连接裕量对拥挤度和延时折中关系的参数。一般取 $\mathrm{MaxCrit}=0.99$，$\eta=1$ 以获得最佳效果。当 $\mathrm{Crit}(i,j)=0$ 时，$\mathrm{Cost}(n)$ 就退化成仅考虑拥挤度的路径搜索算法。

(2) 式中的 $\mathrm{delay}_{\mathrm{Elmore}}$ 是当前扩展的节点 n 及其拓扑结构的函数。为了计算在布线过程中增加一个节点的延时增益，就需要知道对这个节点进行放电的上游电阻 R_{upstream} 的值。通过布线开关 switch 到达节点 m 时节点 n 的上游电阻为

$$R_{\text{upstream}}(n)=\begin{cases}R(\text{switch})+R_{\text{metal}}(n), & \text{如果 switch 是三态缓冲器}\\ R_{\text{upstream}}(m)+R(\text{switch})+R_{\text{metal}}(n), & \text{如果 switch 是传输管}\end{cases} \tag{3-14}$$

其中 $R_{\text{metal}}(n)$表示该布线资源节点的金属电阻，对于非互连线段的其他节点来说其值为 0。对于这种拓扑结构，节点 n 的 Elmore 延时是

$$T_{\text{Elmore}}(n,\text{topology})=T_{d\text{-intrinsic}}(\text{switch})+\left[R_{\text{upstream}}(n)-\frac{R_{\text{metal}}(n)}{2}\right]C_{\text{total}}(n) \tag{3-15}$$

式中，C_{total}是节点 n 的总电容，包含金属电容和寄生开关电容。由于金属电阻是分布式的，因此需要将 $R_{\text{upstream}}(n)$减掉 $R_{\text{metal}}(n)$的一半。

(3) $b(n)\cdot h(n)\cdot p(n)$ 用于平衡布线资源的拥挤度。$b(n)$是节点 n 的基本成本，可以设为它本身的延时值。因为迷宫布线扩展到对应于线网连接的漏端时就终止，所以通过设置合适的布线资源成本，使得迷宫扩展更早抵达漏端，从而达到节约 CPU 运行时间的目的[11]。

$p(n)$是节点 n 的当前拥挤度成本。如果使用这个节点对当前连接布线不会造成任何重用，$p(n)$就为 1；$p(n)$随着节点重用次数的增加而增加。另外 $p(n)$是已运行的布线迭代次数的函数。在初始迭代中，$p(n)$随着当前节点 n 的重用次数缓慢增加；在后期迭代中，$p(n)$随着节点 n 的重用次数快速增加。只要一根线网被拆线重布时就要按下式更新 $p(n)$：

$$p(n)=1+\max(0,[\text{occ}(n)+1-\text{cap}(n)]\cdot p_{\text{fac}}) \tag{3-16}$$

其中 occ(n)表示当前使用节点 n 的线网数目，cap(n)是节点 n 所能有效使用的最大线网数，一般值为 1。$h(n)$是节点 n 的历史拥挤度。每次节点 n 被重复利用，$h(n)$就增加，并且布线器会保持这个“拥挤记录”。只有在一次完整的布线迭代后，历史拥挤度才被更新。第 i 次布线迭代时其值为

$$h(n)^i=\begin{cases}1, & i=1\\ h(n)^{i-1}+\max(0,[\text{occ}(n)-\text{cap}(n)]\cdot h_{\text{fac}}), & i>1\end{cases} \tag{3-17}$$

每一次布线迭代时 h_{fac}和 p_{fac}的不同选择就决定了不同的布线策略。可以通过增加 p_{fac}值来迅速增大 $p(n)$，从而在前期迭代时也会尽量考虑拥挤度因素。

(4) PathCost(n)为从当前部分布线树到节点 n 的总路径成本，即

$$\text{PathCost}(n)=\sum_{l\in \text{path from } RT(r) \text{ to } n}\text{Cost}(l) \tag{3-18}$$

3.4.2 物理综合技术

对于 FPGA 芯片来说，关键路径延时中的绝大部分是互连延时。然而，传统的逻辑综合工具仅仅基于网表结构估算电路中的互连延时，导致逻辑优化后的关键路径延时与布局布线后的情况相比存在明显偏差。

例如，逻辑综合工具会根据网表的连接关系对图 3-21 所示的多扇出线网 1 进行延时估算。但是某些漏端经过布局后有可能与源端位于相同的逻辑单簇内。例如在图中就有一个漏端和源端位于同一个左上角的可编程逻辑簇。由于同一逻辑簇内的互连延时往往要小于逻辑簇之间的互连延时，因此逻辑综合工具所估算这个漏端的互连延时要比布局后的实际情况大很多。假设在逻辑综合时这个漏端位于关键路径上，但是在布局后这个漏端的延时

就会减少，关键路径的延时就会改变，这样就加剧了逻辑综合和布局布线之间的时序收敛(timing closure，有时也译作“时序逼近”)问题。为了解决布局布线前后的延时偏差问题，需要在布局布线后利用物理综合(physical synthesis)技术以便根据器件的实际布局情况准确估算电路的互连延时，从而更好地优化关键路径，提高电路的性能。在专用芯片设计中，针对时序收敛问题的物理综合优化技术主要为门尺寸调整以及缓冲器插入技术等。但是这些技术并不适用于 FPGA，这是因为 FPGA 所有的硬件资源都事先制造的。以下讨论适用于 FPGA 的物理综合技术。

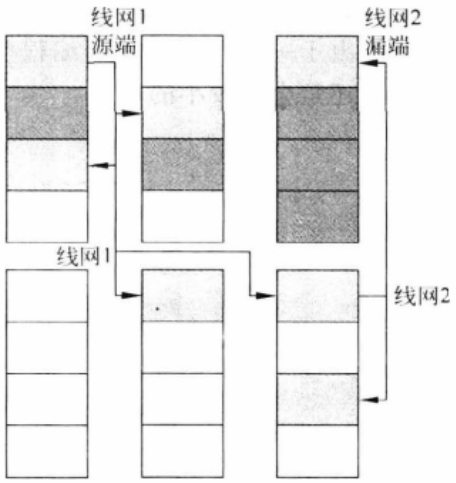
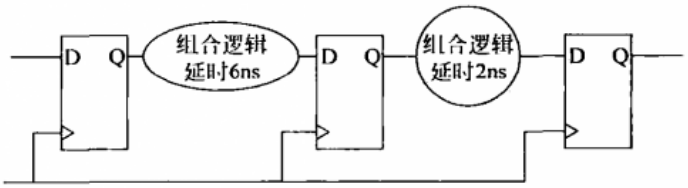


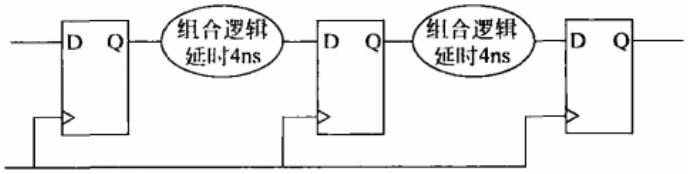
图 3-21　多扇出线网的互连延时估计偏差

1. 寄存器重定时(register retiming)

寄存器重定时是指通过改变触发器或寄存器的位置，如通过减少非关键路径上的延时裕量来增加关键路径上的延时裕量，以便更好地平衡不同时序电路之间的延时。如图 3-22(a)所示，图中关键路径延时是 6ns，而非关键路径延时为 2ns。因此，如果能采取寄存器重定时的办法，把关键路径上的部分延时转移到非关键路径，那么电路的工作频率就会上升。如图 3-22(b)所示，寄存器重定时技术会平衡组合逻辑的延时，使得关键路径延时由原来的 6ns 降低为 4ns。



(a) 重定时前的电路

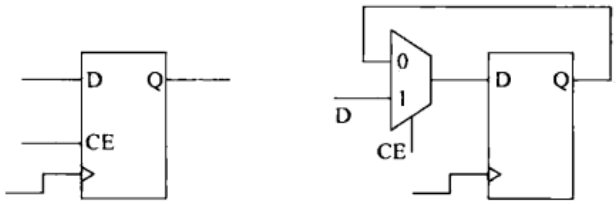


(b) 重定时后的电路

图 3-22　寄存器重定时举例

然而，不是所有的情况都可以取得很好的寄存器重定时效果。有时如果组合逻辑可以再划分，那么可以把关键延时通过插入寄存器的办法进行拆分。有时如果控制信号不兼容，将会导致不能简单地通过改变寄存器位置来实现重定时。这时，就需要分离不兼容的控制信号。如图 3-23 所示，图中的触发器有时钟使能端(CE)，它就不能和后端的不带时钟使能端的组合逻辑延时进行重定时。其中一个办法是可以增加一个数据选择器，当 CE 为 0 时，触发器的输出经过数据选择器反馈到输入，因此即使有时钟输入时输出仍保持不变。只有

当 CE 为 1 时，外部的输入 D 才会连到触发器输入端。这样就实现了带时钟使能端的触发器功能，从而可以进行寄存器重定时。不过此时前级的组合逻辑就增加了一个数据选择器的延时。由于一般的 FPGA 可编程单元中都会有一些没有用到的资源，因此这个数据选择器的延时还是会比较小的。



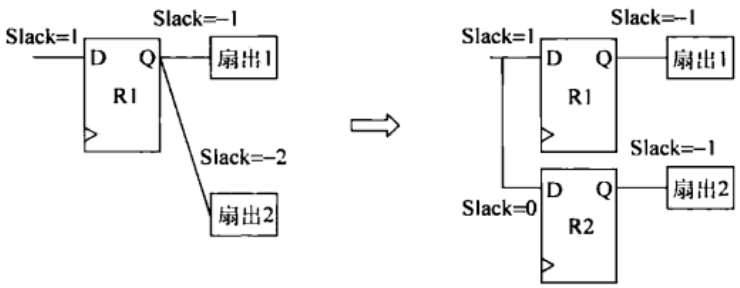
(a) 带时钟使能输入的触发器　(b) 转化为不带时钟使能输入的触发器

图 3-23　分离不兼容的控制信号以实现寄存器重定时

2. 寄存器复制(register duplication)

一般而言，在电路的网表结构中，寄存器输出普遍是多扇出的。FPGA 包含有大量的寄存器，这些寄存器对大部分电路来说不会全部被使用。因此通过寄存器复制的方法，就可以充分利用这些空闲的寄存器以提高电路的时序性能。

如图 3-24(a)所示，触发器 R1 驱动一个线网，它共有两个扇出，延时裕量均为负值。这时，可以把触发器 R1 复制到寄存器 R2，或者说是在电路中增加了一个触发器，并且将 R2 放置得靠近扇出 2。这时相当于将 R2 输入端 D 的部分延时裕量移至扇出 2 的路径上，使得 R2 的 D 端延时裕量下降为 0，而连接 2 的路径上的延时裕量改善为 −1，如图 3-24(b)所示。



(a) 寄存器复制前的时序结果　(b) 寄存器复制后的时序结果

图 3-24　寄存器复制方法示例

又如在图 3-21 中，可编程逻辑簇中涂色的基本可编程逻辑单元都已经被占用了，而没有涂色的基本可编程逻辑单元是空闲的。假设线网 1 的源端经过右下角的逻辑簇再到达线网 2 的漏端后构成了电路的关键路径，可以发现如果把右下角的逻辑簇内用到的基本逻辑单元拷贝到离线网 2 漏端很近的空闲基本可编程单元，即图 3-25 中所标记的“拷贝的逻辑单元”，那么关键路径的延时就会明显降低。可以发现此时的寄存器复制实际上是连这个寄存器所在的可编程逻辑单元一起拷贝过去了，包含它所连接的组合逻辑。经过寄存器复制后的电路还需要运行局部布局布线以得到正确的输出。不过物理综合技术工具会自动完成局部的布局布线任务。

3. 逻辑重构(logic reconstructing)

上面介绍的两种物理综合技术，不管是寄存器重定时还是寄存器复制，都要增加电路的硬件资源成本。逻辑重构的方法并不需要增减器件，它只需在保证逻辑功能等价的前提下交换各逻辑单元之间的连接以达到优化时序性能的目的。

考虑一个由两级 4 输入 LUT 实现的 10 输入与非门电路，如图 3-26 所示。第一级中存在一个延时裕量为－1 的输入信号；第二级 LUT 中有两个延时裕量为＋2 的输入。这时，延时裕量为－1 的输入信号可以与第二级中的一个延时裕量为＋2 的输入信号进行交换，交换后的电路就不存在延时裕量为负的路径了。同时我们可以更新 LUT 的码点来保证逻辑功能等价性。

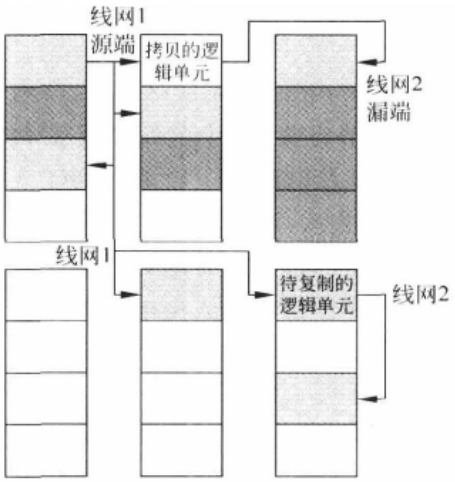


图 3-25　寄存器及其逻辑单元复制

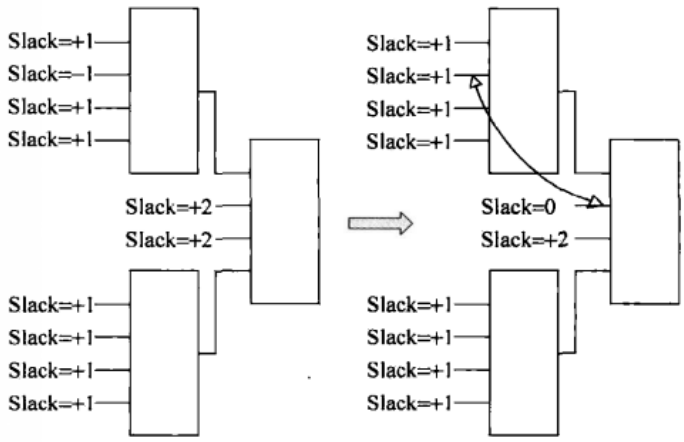


图 3-26　逻辑重构举例

3.5　时序分析

数字电路设计一般包含组合逻辑和时序单元两部分，其中电路的时序性能主要取决于组合逻辑的延时。如图 3-3 所示，数字电路的设计流程中均需要仿真工具来验证电路的功能和时序关系。但是由于激励向量覆盖率的限制，只借助于仿真方法的电路功能和时序性能得不到完全保证。要使电路正确工作就必须使电路按照预先明确定义好的时序关系进行工作，否则可能会导致错误的数据写入触发器或寄存器内部的储存单元。静态时序分析就是基于电路的拓扑结构，计算逻辑单元延时和互连延时，并对电路进行延时分析，找出影响电路时序的关键路径，确定电路能够工作的最高速度。

3.5.1　动态时序仿真和静态时序分析

动态时序仿真(dynamic timing simulation)采用“事件驱动”的方法，通过在电路上加一系列随时间变化的激励向量来计算电路的时序行为。动态时序仿真的特点在于用户在编写输入激励时会受到电路结构的限制，并且可能需要在不同的时钟频率下进行模拟和仿真，所耗费的时间长、软件运行速度慢。对于大规模电路来说，用户很难提供完备的激励文件，因

此验证不充分。动态时序仿真非常适合于在给定的输入下对电路进行分析和查错。

静态时序分析(static timing analysis)是一种不需要输入激励的时序性能分析技术。其作用是分析电路时序的最坏情况,验证此时的电路性能是否能满足用户的要求,这样就可以确保在任何情况下电路都能正常可靠地工作。由于它不需要任何激励信号,因此速度快、验证充分。它能够找出电路中发生时序冲突(timing violation)的各个路径,即不符合用户时序需求的路径。但正是因为没有激励,因此无法获取电路所有的功能信息,以至于会分析实际并不存在的"伪路径"(false path)。而且静态时序分析只能分析同步时序电路,对异步时序电路的时序分析还不够成熟。对于大规模的数字电路来说,静态时序分析方法要比动态时序仿真在运算效率和时序检查的覆盖率方面有明显的优势。因此本节后续内容将主要介绍静态时序分析。

3.5.2 时序图

在进行静态时序分析时,首先要将具体电路转换成时序图(timing graph)。任意一个组合或时序逻辑电路都可以用时序图 G=(V,E)表示,其中 V 为节点集,V 中的每个元素包括电路中的主输入(primary input)和主输出(primary output)以及电路中逻辑单元的输入输出端口。如果节点集 V 中的两个节点 u 和 v 在电路中有连接关系,那么就会被 E 中的有向边 $e(u,v)$连接。在组合逻辑电路中,时序图的主输入就是电路的输入;主输出就是电路的输出,其他节点表示逻辑单元的端口。在时序逻辑电路中,时序图的主输入由电路的输入和寄存器单元的时钟输入端组成。主输出由电路的输出和寄存器单元除时钟输入端以外的其余输入端组成。

图 3-27 所示为一个包含五个输入、三个输出、若干个组合逻辑门以及一个触发器的数字电路。时序图转换过程如图 3-28 所示。图中节点内的标号表示各个逻辑单元的端口名字,其中五个输入端对应于时序图中的五个主输入节点,三个输出端和触发器的输入端 D 构成了四个主输出节点,其余的逻辑单元输入输出端分别对应于一个内部节点。由于触发器的输出需要时钟的触发,因此在触发器 DFF 的时钟输入端和它的输出端之间有一条边,从而可以对输出延时进行时序分析。

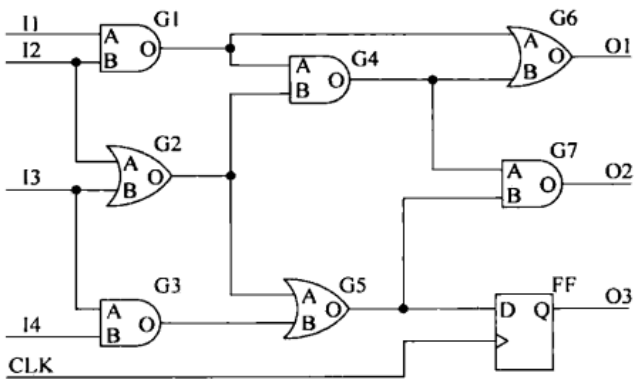


图 3-27 用于举例说明时序图的简单数字电路

通过上述转换得到的电路时序图,可以将电路的拓扑结构很直观地表示出来,从而简化了计算延时以及查找关键路径的难度。接下来讨论如何计算路径延时并确定电路的关键路径。

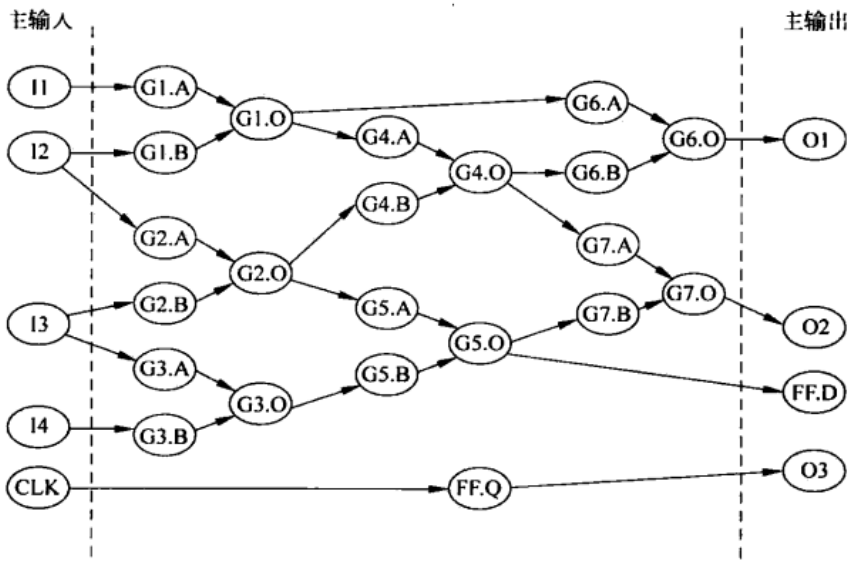


图 3-28　时序图举例

3.5.3　延时计算

如 3.5.1 节所述，静态时序分析不依赖于输入激励而直接根据连接关系找出电路在所有可能的输入组合下最坏情况的延时。静态时序分析的延时计算可以分为基于路径(path-based)和基于块(block-based)的两种方法。

基于路径的计算方法按照深度优先的搜索次序遍历电路中所有可能传播信号的路径，并在遍历过程中计算每条路径的延时，最后统计出在主输出产生最大延时的路径。基于路径计算时序分析方法的优点是可以精确算出每条路径的延时。它的缺陷是占用系统资源大，并且路径的数量随着电路规模急剧上升，因此不适合快速、高效的静态时序分析。

基于块计算的方法使用广度优先的搜索次序来遍历时序图中所有节点，在访问节点时计算该节点的延时，如到达时间、要求时间和时间裕量等参数，从而确定电路是否满足时序要求。其复杂度随电路规模呈线性变化，具有快速、高效的特点，因此本节将对该方法进行重点介绍。关键路径算法(critical path method，CPM)是基于块计算方法中最具代表性的一种算法。现在 CPM 已经集成到大多数电路延时计算快速算法中。图 3-29 所示为 CPM 算法的伪代码[12]。

如图 3-29 的伪代码所示，CPM 遍历并计算电路时序图中每个节点的延时，主要经过以下三个步骤。

步骤一：将所有节点的输入访问个数全部设置为 0，即图 3-29 中的第 2 至 3 行。

步骤二：对于所有主输入节点的扇出节点，全部加入待处理队列 Q，即图 3-29 中的第 4 至 7 行。

步骤三：若队列 Q 非空，取出 Q 的队首元素 g，计算 g 的延时，之后查找 g 所有的扇出节点，若扇出节点的输入访问个数等于其输入个数，则说明其扇入节点已经全部被访问过，将其加入待处理队列 Q。重复步骤三直至 Q 为空。这个步骤对应于图 3-29 中的第 8 至 14 行。

```
procedure CRITICAL_PATH_METHOD
1   Q = ∅;
2   for all vertices i ∈ V
3         n_visited_inputs[i] = 0;
      /* Add a vertex to the tail of Q if all inputs are ready */
4   for all primary inputs i
5       /* Fanout gates of i */
6       for all vertices j such that (i→j) ∈ E
7           if ( ++n_visited_inputs[j] == n_inputs[j]) addQ(j,Q);
8   while (Q ≠ ∅) {
9       g = top(Q);
10      remove(g,Q);
11      compute_delay[g];
12      /* Fanout gates of g */
13      for all vertices k such that (g→k) ∈ E
14          if ( ++n_visited_inputs[k] == n_inputs[k]) addQ(k,Q);
}
```

图 3-29 CPM 算法的伪代码

现在以图 3-30 为例说明 CPM 算法中计算节点的“到达时间”(arrival time)过程。图中节点内部的号码为节点标号,“*D*”表示该节点的输入输出之间的延时,方框中“*A*”的数据为节点的到达时间。

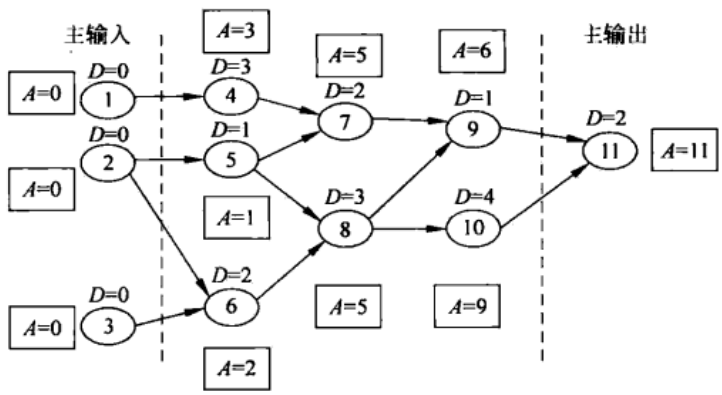


图 3-30 利用 CPM 算法计算电路的到达时间

假设上图中主输入的到达时间均为 0,则通过 CPM 算法遍历电路时序图并计算电路延时的过程如下。

(1) 节点 4 入队。

(2) 节点 5 入队。

(3) 节点 6 入队。

(4) 节点 4 出队,计算节点 4 的到达时间,为节点 1 的到达时间(0)加上节点 4 本身的延时(3),得到节点 4 的到达时间为 0+3=3。

(5) 节点 5 出队,计算节点 5 的到达时间(计算过程同节点 4),节点 7 入队。

(6) 节点 6 出队,计算节点 6 的到达时间(计算过程同节点 4),节点 8 入队。

(7) 节点 7 出队,计算节点 7 的到达时间。由于节点 7 有两个前级节点(节点 4 及节点

5)，首先比较两个前级节点到达时间的大小，将其中较大的一个(节点 4 的到达时间(3))加上节点 7 本身的延时(2)，得到节点 7 的到达时间为 5。

(8) 节点 8 出队，计算节点 8 的到达时间(计算过程同节点 7)，节点 9 入队，节点 10 入队。

(9) 节点 9 出队，计算节点 9 的到达时间(计算过程同节点 4)。

(10) 节点 10 出队，计算节点 10 的到达时间(计算过程同节点 4)，节点 11 入队。

(11) 节点 11 出队，计算节点 11 的到达时间(计算过程同节点 4)。

(12) 队列为空，计算结束。

3.5.4　关键路径

与 3.4 节的时序驱动布局布线中的延时信息相比，虽然关键路径的基本概念是一样的，但是静态时序分析对延时的精度要求更高，也需要较长的时间进行计算和分析。而在布局布线过程中，由于经常要调用时序引擎(timing engine)，因此就要求延时的计算和比较在尽可能短的时间内完成，目的是能够快速得出延时变好还是变差的相对效果。它并不需要对延时的绝对数值非常精确，但是独立于布局布线外面的静态时序分析工具就要求延时数据和电路的实际测量数据误差很小，相关的基本概念定义如下。

(1) 到达时间(arrival time，AT)：信号到达该节点时的延时。

(2) 要求时间(required time，RT)：信号被要求在到达该节点时的延时。

(3) 时间裕量(slack)：要求时间减去到达时间，即 Slack＝RT－AT。

关键路径是输入和输出之间延时最大的路径。这些输入输出并不局限于主输入和主输出，它也包括寄存器的输出和下级寄存器输入之间的延时。为了找出关键路径，首先要计算每个节点的到达时间。在计算节点的到达时间时，可以用 CPM 遍历时序图的拓扑结构。在计算节点延时(compute_delay)时，即图 3-29 中的第 11 行，找到该节点的各扇入节点中最大的到达时间，加上该节点本身的延时，得到该节点的到达时间。这样在输出节点处可以很容易找到最大的到达时间。然后再往输入方向搜索，找到最大到达时间所对应的扇入节点，这样一直回溯到某一个输入。这个输入和最大到达时间的输出节点之间的路径就是关键路径。

找到关键路径后就可以利用“要求时间”来检查电路的时序约束是否满足。如果用户没有对“要求时间”设置约束，那么就可以用关键路径延时作为要求，采用从输出往输入方向反向遍历时序图的方法，依次得到各个节点的要求时间，再用要求时间减去到达时间，得到时间裕量。如果时间裕量小于零，则说明时序要求不满足，该部分电路需要优化加速；如果时间裕量大于零，则说明时序要求满足，设计者还可以降低该部分速度以节省硬件资源。

在图 3-30 的时序图中已经计算出每个节点的到达时间，下面再计算每个节点的要求时间和时间裕量。如图 3-31 所示，方框中 A 的值表示到达时间，R 表示要求时间，S 表示时间裕量。从计算结果可以看出，节点标号次序为 3→6→8→10→11 这条路径构成了这个电路的关键路径。假设节点 11 的要求时间约束为 10，则时间裕量为－1。这一结果表明这个电路不符合时序约束要求，需要对该条路径进行优化加速。而在 1→4→7→9→11 这条路径上，除节点 11 的时间裕量小于零之外，其余节点的时间裕量均大于零，说明这条路径满足时序约束要求，可适当放慢该条路径的速度以节省电路的硬件资源。

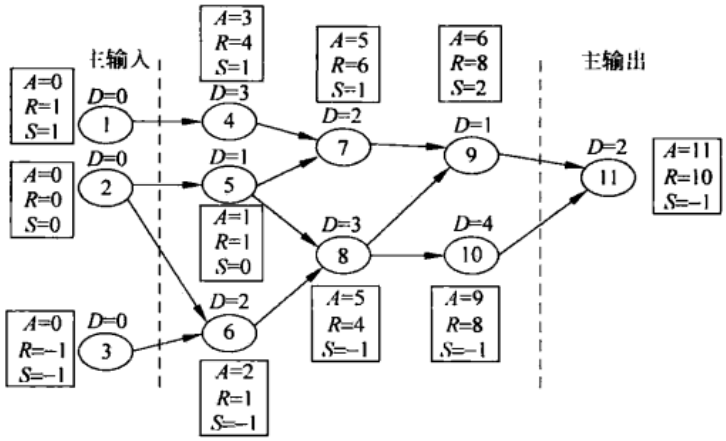


图 3-31 利用 CPM 算法计算电路的要求时间和时间裕量

3.5.5 建立时间和保持时间检查与分析

在时序电路中，触发器作为时序单元以隔开各组合逻辑电路的输入输出，是电路流水线工作的基本元件。因此为保证时序电路的正常工作，必须对触发器的时序行为进行分析，保证它能够在前后级组合电路之间正常地传递数据。由触发器的工作原理可以知道，在触发器时钟的有效边沿时，触发器的输入端数据就被保存到内部存储单元并且触发器内部保存的数据就会传送到触发器的输出端。因此，为了采集到有效的数据并传送到输出端，触发器的输入数据必须在时钟触发边沿前后的一定持续时间内保持稳定，否则触发器保存的数据就不会稳定，从而影响电路的正常工作。数据在触发器时钟边沿之前和之后所要求保持稳定的时间分别称做建立时间和保持时间。两者的要求时间和实际时间的差别同样记为时间裕量(slack)。如果建立时间或保持时间的时间裕量为正，则表示相应的条件满足；如果值为负，则表示条件不满足，触发器不能正常工作；如果时间裕量为 0，则表示条件刚好满足。

建立时间(setup time，记为 t_{su})是触发器在时钟边沿到达之前，其数据输入端的数据必须保持稳定不变的时间。如果在这段时间内输入端数据还在变化，那么触发器内部存储的数据就不能被正常地“建立”起来。因此，从建立时间的要求来说，它是对输入端数据的延时提出要求，即在触发器时钟边沿到达前，数据必须要先到达触发器的数据输入端。换句话说，建立时间主要是检查输入端数据不能来得太慢——在下个有效时钟边沿前，数据端的组合电路延时不能太大，一定要事先建立好。电路的最高时序频率主要由建立时间条件限制。如果组合逻辑电路延时很小，那么建立时间条件就很容易满足，电路的工作频率就很高。如果组合逻辑电路延时很大，那么建立时间条件就不容易满足，电路的工作频率就比较低。为了提高电路的工作频率，就要尽量减少组合电路的延时，保证建立时间条件得到满足。

保持时间(hold time，记为 t_{hold})是触发器在时钟边沿来到后，其数据输入端的数据必须保持不变的时间。如果在这段时间内输入端数据还在变化，那么触发器内部存储的数据就可能无法“保持”住。因此，从保持时间的要求来说，它同样是对输入端数据的延时提出要求，即在触发器时钟边沿到达后，输入端数据必须要保持住，不能改变。换句话说，保持时间

主要是检查输入端数据不能变化太快——在本次有效时钟边沿后，数据端的组合电路延时不能太小，一定要保持不变。如果组合逻辑电路延时很小，那么保持时间条件就不容易满足，需要在组合电路插入缓冲器来增大组合逻辑延时。如果组合逻辑电路延时很大，那么保持时间条件就容易满足。如果触发器的保持时间条件为0，那么这表示时钟边沿后输入数据就允许变动，而不会影响内部存储单元的有效数据。因此这个条件是比较容易满足的。图 3-32 表示了触发器的建立时间和保持时间。

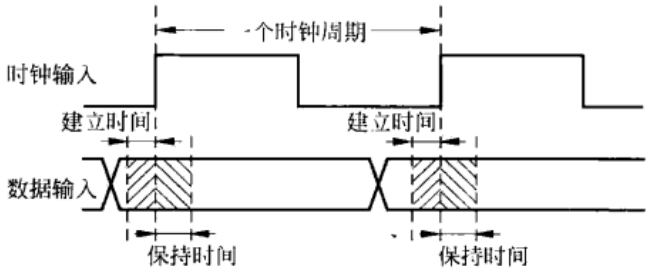


图 3-32　触发器的建立时间和保持时间

以上是对建立时间和保持时间作定性分析。为了能够定量地衡量这两个触发器正常工作的基本条件，就需要引入针对建立时间和保持时间的时间裕量计算方法。如图 3-33 所示，同步数据通路的电路参数表示如下。

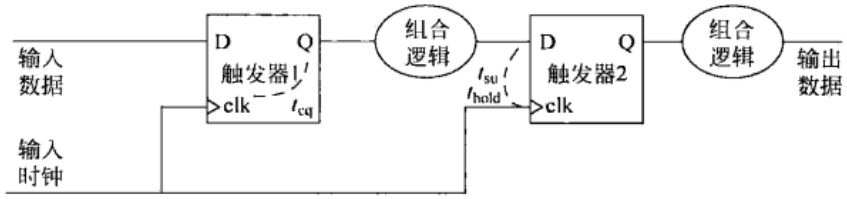


图 3-33　数据通路的电路和时序参数

t_{clk}：时钟周期。

t_{cq}：触发器从输入时钟端的有效边沿触发到输出端 Q 有稳定输出所需要的延时。

t_{su}：由触发器本身的电路属性所决定的建立时间。

t_{hold}：由触发器本身的电路属性所决定的保持时间。

t_{logic}：组合逻辑的延时。

t_{skew}：时钟到达触发器 2 的时钟端和触发器 1 的时钟端的时钟偏差。一般来说，时钟到达触发器 2 的时钟端要比到达触发器 1 的时钟端慢，两者之间的时差就是 t_{skew}。下面定量分析输入数据在触发器 1 的上边沿时钟触发下，经过中间的组合电路后输出到触发器 2 的输入 D 端的延时和触发器 2 时钟输入端的时序关系。

在触发器 1 输入时钟上边沿的触发下，不妨设此时为 0 时刻，触发器 1 的 Q 端在经过 t_{cq} 的时间后就有了稳定的输出，然后再经过中间的组合电路延时后到达下一级触发器 2 的输入端 D。最后到达触发器 2 后要有 t_{su} 的时间保持不变，这样才能被下一个时钟边沿正确获取。因此数据从前一级的触发边沿到达下一级触发器的时钟触发边沿所经过的总延时共为 $t_{cq}+t_{logic}+t_{su}$。从时钟输入来说，它从上一个触发器的边沿到达下一个触发器的边沿共

需要的时间为 $t_{clk}+t_{skew}$。这里我们假设默认情况下，下一个时钟边沿就是一个时钟周期后的下一级触发器的有效边沿。对于多时钟节拍路径(multi-cycle path)的时序约束来说，组合逻辑就需要一个以上的时钟周期才能完成计算。建立时间条件要求数据延时不能超过时钟时间，即有下式成立：

$$t_{cq}+t_{logic}+t_{su}\leqslant t_{clk}+t_{skew} \tag{3-19}$$

由于时间裕量 Slack 为正数时表示时序条件满足，因此建立时间的时间裕量 $Slack_{su}$ 可以定义为

$$Slack_{su}=t_{clk}+t_{skew}-t_{cq}-t_{logic}-t_{su} \tag{3-20}$$

通过式(3-20)就可以定量地衡量建立时间在多大程度上得到满足。下面再按照同样的方法推导保持时间的时间裕量 $Slack_{hold}$ 表达式。

保持时间条件主要是检查数据通路变化不能太快，即数据一定要保持住。由于 t_{hold} 一般不会小于零，因此 t_{hold} 值就加到时钟的延时，用以约束时钟延时不能太慢。换句话说，在有效数据变化前，时钟有效边沿一定要到达。假设在 0 时刻，触发器 1 在时钟边沿的触发下数据传到下一级触发器的 D 输入端延时为 $t_{cq}+t_{logic}$。这里 t_{hold} 并没有加到数据延时。反之，t_{hold} 是加到时钟的延时，用来约束时钟不能太慢。因此时钟在两个触发器之间的延时为 $t_{skew}+t_{hold}$。保持条件要求数据延时不能小于时钟延时，即有下式成立：

$$t_{skew}+t_{hold}\leqslant t_{cq}+t_{logic} \tag{3-21}$$

由于时间裕量 Slack 为正数时表示时序条件满足，因此保持时间的时间裕量 $Slack_{hold}$ 可以定义为

$$Slack_{hold}=t_{cq}+t_{logic}-t_{skew}-t_{hold} \tag{3-22}$$

通过式(3-22)就可以定量地衡量保持时间在多大程度上得到满足。由式(3-22)可知，一般电路的时钟树偏差是很小的，即 t_{skew} 接近于 0。并且触发器的 t_{hold} 一般也接近于 0 或等于 0。因此式(3-22)是非常容易满足的，即组合逻辑的延时一般要远远大于时钟树各个端点的时差。保持时间条件不满足的情况一般发生于非时钟树上的普通互连资源时钟信号，例如异步时钟或者门控时钟等。此时组合逻辑延时会小于前后级触发器的时钟偏差，从而导致保持时间冲突。最近由于半导体工艺进入纳米尺度，不同的芯片之间存在比较大的工艺偏差(process variation)，因此为了保证时序电路的稳定工作，通常还要引入一定的建立时间不确定值(clock setup uncertainty)和保持时间不确定值(clock hold uncertainty)，以进一步约束建立时间和保持时间条件，保证电路即使在考虑工艺偏差的情况下也能在时序分析工具所报告的最高频率下稳定工作。

3.6 基于 JTAG 的在线分析技术

本章前面各节所描述的电路仿真，时序分析与优化技术是常用的软件算法，它们是通过软件建模和计算、评估的方法间接地衡量电路的性能。但是在实际应用中，软件方法毕竟和硬件的实际工作状态可能会存在差别，并且软件运行速度一般都远远慢于芯片的实际运行速度，因此本节介绍直接把 FPGA 内部电路的实时信息利用 JTAG 技术进行在线分析的方法。

出于对复杂芯片的快速测试需求，20 世纪 80 年代提出了基于边界扫描的自动化测试

技术；它的基本思想就是在芯片四周的输入输出端口上增加一圈移位寄存器单元，即边界扫描寄存器链。这些边界扫描寄存器可以把芯片内部的在线寄存器信息保存起来，然后通过一定的时序控制把这些信息利用极少的管脚传到芯片外面供用户进行测试分析。边界扫描测试已经成为数字系统可测性设计的主流技术。1990 年美国 IEEE 组织确定了 IEEE 1149.1 标准。JTAG 是联合测试行动小组(Joint Test Action Group)的简称，也是基于这个标准结构运行测试机制的常用叫法。虽然 JTAG 技术起源于电路的自动测试需求，但是这个标准也很容易用于 FPGA 电路的在线分析。

3.6.1　JTAG 基本结构和原理

根据 IEEE 1149.1 标准规定，JTAG 硬件结构包括以下四个部分。

1. 测试访问端口(test acess port，TAP)

测试访问端口是 JTAG 对外通信的端口。考虑到 JTAG 端口数量的限制，一般边界扫描设计中 JTAG 只有以下四个非常简单却又很灵活的串行端口。它们的名字都是以字母“T”开头，表示它们是“测试”(test)端口。

(1) 测试时钟输入(test clock，TCK)

时钟信号是时序控制所必需的同步信号。TCK 可以和电路的工作时钟独立。为了提高数据传输效率，一般 TCK 的上升沿用于读取数据，下降沿输出数据。

(2) 串行测试数据输入(test data input，TDI)和串行测试数据输出(test data output，TDO)

这两个串行端口就是 JTAG 唯一的数据通路。不管是指令值，还是用户激励或者是芯片内部的状态值都通过这两个端口进行移位传输。指令值就在控制逻辑的作用下送到 JTAG 内部的指令寄存器。其他数据也都有相应的数据寄存器，例如旁路寄存器、器件标识寄存器和数目众多的边界扫描寄存器等。

(3) 测试模式选择输入(test mode select，TMS)

TDI 和 TDO 所传输的数据有不同的类型，并且也有不同的操作，因此就需要 TMS 输入信号来配合 TAP 控制器从而对数据进行正确的处理。由于 TMS 只是一位控制信号，因此 TMS 输入 0 或者 1 时所表示的意义还和 TAP 控制器当前的状态相关。同时 TAP 控制器也会根据 TMS 的不同输入值跳转到相应的次态，因此 TMS 在 TCK 的触发下，由 TAP 控制器来处理 TDI 和 TDO 数据链上的数据，包括读取、输出或移位等。同时 TAP 控制器本身也受 TMS 的控制，在不同的状态之间跳转，这样这个边界扫描系统就能有序地工作。

需要说明的是，JTAG 还有一个可选的输入端口，用于快速复位 TAP 控制器。但是由于 TAP 控制器不管在什么状态下，只要 TMS 连续输入 5 个“1”，它就会跳到复位状态，因此为了节约 IO 端口的数量，一般都不使用这个复位端口。

2. TAP 控制器

在介绍 TAP 测试端口的时候，我们知道 JTAG 扫描链需要外部 TMS 输入信号对 JTAG 的内部状态机进行控制才能协调工作。图 3-34 就是 IEEE 1149.1 标准所规定的 TAP 控制器的状态图。它共有 16 个状态，除了图中左上角的两个“复位”(Test-Logic-

Reset)和“运行/空闲”(Run-test/idle)状态外,其余的14个状态被分为两组。每组均有7个状态,分别控制数据寄存器和指令寄存器。箭头边上的“0”或“1”就表示TMS的输入值。可以发现不管当前为何种状态,只要TMS连续输入5个“1”,那么状态机就会自动跳到左上角的“复位”态。

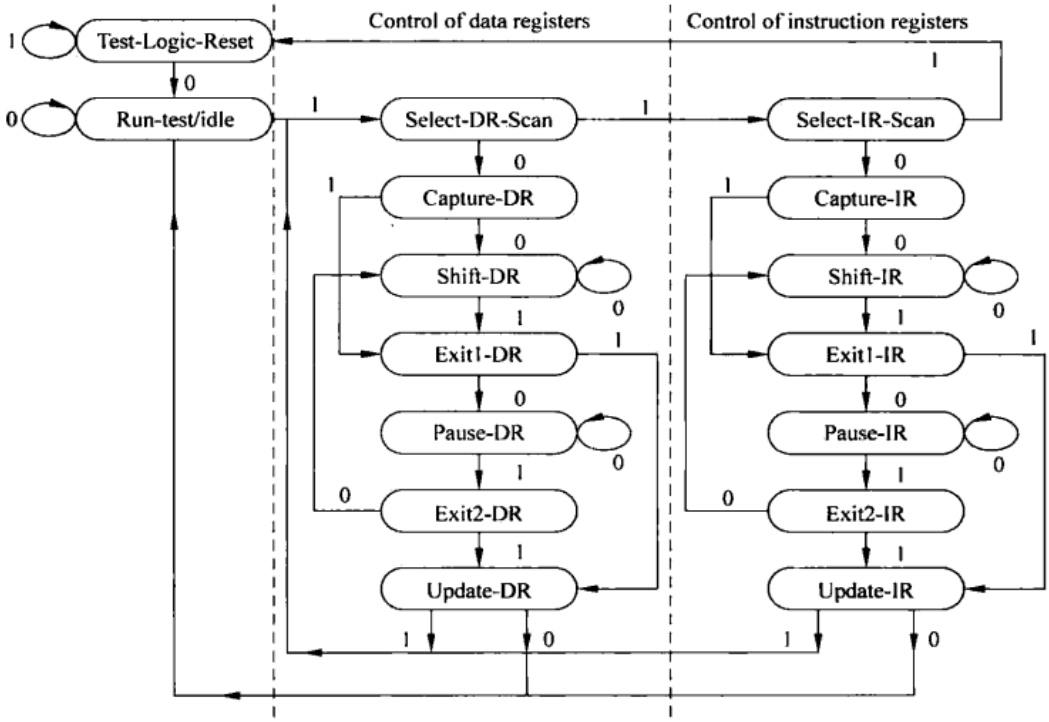


图3-34 TAP控制器的状态图

3. 指令寄存器(instruction register,IR)

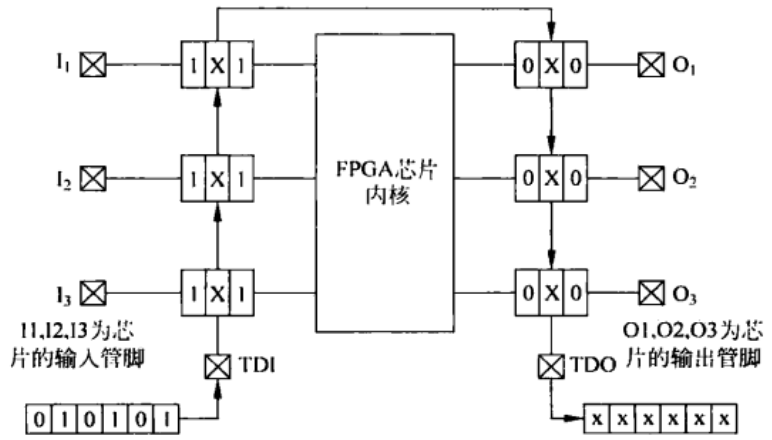
指令寄存器用于存储用户输入的指令码,它的长度等于指令的长度,一般为4位。这样一共可以有16条不同的指令。如果指令条数不够,就可以增加指令寄存器的长度。JTAG运行时指令码经过TDI移入到指令移位寄存器,然后再经过指令译码,产生控制边界扫描电路的各种信号。这些译码逻辑电路是由指令译码器来完成的。

4. 测试数据寄存器(test data register,TDR)

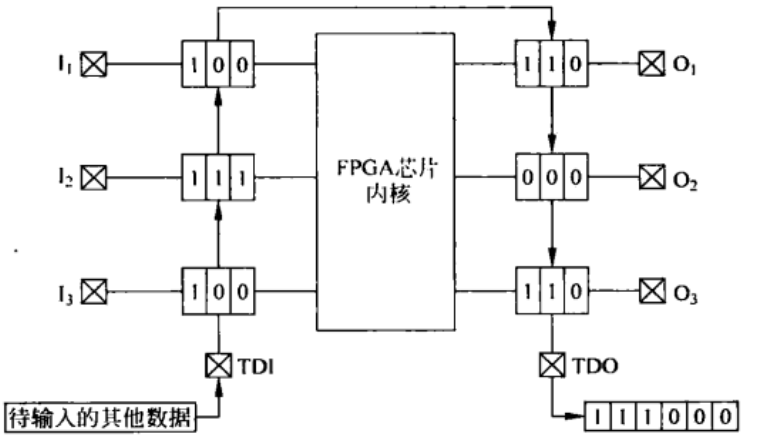
除了指令寄存器外,JTAG内部还有很多的数据寄存器。常用的数据寄存器包含旁路寄存器、器件标识寄存器、边界扫描寄存器组等。这些都可以接入边界扫描链,接受从TDI输入的信号,同时把寄存器的数据从TDO端口输出。在一个确定的时刻寄存器的数据选择则由指令寄存器中的值控制。

图3-35是基本的JTAG工作原理示意图。假设FPGA芯片内核的用户电路一共有三个输入端口和三个输出端口。图中只画出了JTAG链的两个数据端口:TDI和TDO。TMS和TCK端口没有画出,它们主要是用于控制JTAG数据链的各种操作和同步。JTAG边界扫描链把芯片内核和芯片外部的管脚I1、I2、I3和O1、O2、O3隔离开来。当

JTAG 不工作时，芯片内核和外部管脚是直连的。因此 JTAG 的引入并不影响芯片的正常工作。下面简单举例说明 JTAG 的基本工作流程。我们假设外部数据为“101010”，它们要经过 TDI 端口送给芯片内核作为输入激励。另外还要把芯片内核的端口数据“111000”(顺时针方向)从 TDO 输出到外部，供用户检查分析。如图 3-35(a)所示，一开始边界扫描链中的值是随机的，在图中用“X”表示。先从 TMS 连续输入 5 个“1”，使状态机进入“复位”态。然后 TMS 输入“01100”，从图 3-34 的状态图可知状态机依次进入“运行/空闲”、“数据寄存器扫描选择(Select-DR-Scan)”、“指令寄存器扫描选择(Select-IR-Scan)”、“指令寄存器捕获(Capture-IR)”、“指令寄存器移位(Shift-IR)”状态。然后 TMS 输入保持为 0，表示状态机一直处于“指令寄存器移位”状态。这可以从图 3-34 的状态机行为进行确认。此时在 TCK 的作用下，从 TDI 串行输入的数据就会放到指令寄存器中去。JTAG 的指令一般为 4 位数据。IEEE 标准定义了 3 条基本指令。其中包括“旁路(BYPASS)”指令，指令码为“1111”。因此如果要执行旁路指令，那么就从 TDI 在四个 TCK 时钟周期的驱动下输入四个“1”。旁路指令的结果就是旁路掉本芯片中所有的边界扫描链中的边界扫描寄存器，把 TDI 和 TDO 端口直连，从而可以验证 JTAG 控制器是否基本上能够工作，并且可以快速测试同一



(a) 运行前状态



(b) 运行后状态

图 3-35　JTAG 工作原理示意图

个测试板上其余芯片的功能。这样四个时钟周期输入用户所需要执行的指令码后，TMS 就输入“1110”。状态机在四个 TCK 时钟的触发下依次进入“指令寄存器退出(Exit1-IR)”、“指令寄存器更新(Update-IR)”、“数据寄存器扫描选择”、“数据寄存器捕获(Capture-DR)”状态，“数据寄存器捕获”表示以下将对数据寄存器进行操作。具体选择哪个数据寄存器是由指令码经过译码后来选择。假设我们现在所选择的数据寄存器是边界扫描寄存器，因此“数据寄存器捕获”状态表示把芯片管脚上的输出信号值在一个 TCK 时钟的触发下全部捕获到相应的边界扫描移位寄存器。在此以前边界扫描移位寄存器中的值可能是随机的。经过捕获后这些寄存器就保存了当前芯片 IO 端口的信号值。然后 TMS 端口输入一个“0”，这样状态机就跳转到“数据寄存器移位(Shift-DR)”状态。接下去 TMS 就保持为“0”值，状态机一直停留在“数据寄存器移位”状态。从 TDI 输入的数据就会移位到边界扫描寄存器单元。在边界扫描链移位的过程中，TDO 就把刚才捕获的当前芯片 IO 信号值串行输出。经过六个时钟后，这些边界扫描寄存器都从 TDI 得到了相应的值。然后 TMS 端输入“11”，状态机就依次进入“数据寄存器退出(Exit1-DR)”、“数据寄存器更新(Update-DR)”状态。在这个状态下只需要一个时钟就会把从 TDI 读入的边界扫描寄存器中的值分别传给芯片的输入 IO 端口作为激励。运行后 JTAG 的数据值如图 3-35(b)所示。由图可见 FPGA 内核的各端口已经加上了由外部经过扫描链而输入的激励，但是扫描链外圈的芯片管脚的信号值，即连接于 I1、I2、I3 和 O1、O2、O3 的信号值并没有发生变化。

3.6.2 基于 JTAG 软扫描链的在线分析方法

虽然 JTAG 最初是为了芯片测试而提出的规范，但是由于它接口非常简单，一位数据线既可以传送指令码，又可以传送数据，因此 JTAG 标准完全可以用于芯片的在线测试，即利用 JTAG 标准可以把 FPGA 内部的信号拉到 IO 端口，然后再通过 JTAG 指令对这些芯片内部的信号进行在线分析和检查。另外，FPGA 还能提供另一种实现方法。我们可以利用 FPGA 的可编程资源来实现 JTAG 电路，即设计一个 JTAG 软扫描链，把它加到用户电路的周围，并把所需要测试或分析的芯片内部信号都拉到 JTAG 软扫描链中的边界扫描单元进行在线分析或者波形显示。然后用户只要编写软件代码就可以控制 JTAG 的操作方式，其中包括向用户电路施加激励，读取电路的实时运行结果，并利用图形界面进行波形显示。JTAG 软扫描链的另一个好处就是可以根据待测的信号数目调整边界扫描链的长短。FPGA 芯片内部 JTAG 硬核所控制的扫描链长度一般连到所有的 IO 端口，因此扫描链长度一般都比较长。缩短边界扫描链的长度能够减少扫描时钟的节拍数，加快芯片的在线分析速度。商用 FPGA 供应商一般都提供了很方便的在线测试工具。它们一般把用户所需要的数据保存到 FPGA 片内没有用到的嵌入式存储器中，然后再通过软件读出这些数据，并通过图形界面显示出来。但是通过练习使用 JTAG 软扫描链的方法可以很好地掌握 FPGA 硬件电路设计及其软件接口，对于更好地掌握 FPGA 的底层结构和软硬件协同设计方法很有好处。以下我们介绍基于 JTAG 软扫描链的软件部分设计方法。

由于 JTAG 端口部分很简单，并且它的控制指令和数据都可以从 TDI、TDO 端口读写，因此只要利用电脑上的端口和 FPGA 器件内部的 TAP 控制器硬核，就能够通过电脑端口通信连接 JTAG 的四个端口，从而可以编写软件代码进行 FPGA 电路信号的在线分析了。

以一个简单的四位计数器 counter4 为例，假设设计文件名为 counter4. v。对应的 Verilog

代码如图 3-36 所示。

```
module counter4(clk,clr,out);
input clk,clr;
output[3:0] out;
reg[3:0] out;

always @(posedge clk or posedge clr)
begin
if (clr)   out<=0;
else       out<=out+1;
end
endmodule
```

图 3-36　用于 JTAG 软扫描链进行在线测试的 counter4 电路描述

我们可以编写一个程序，它能自动读入图 3-36 中的 Verilog 文件，并提取输入输出管脚信息。counter4.v 中有两个输入管脚，四个输出管脚。注意到电路输出 out 为四位的总线形式，为了方便统一处理，需要将它拆分成 out[0]～out[3]四个单独的输出，分别连到独立的 IO 单元进行控制。获得输入输出管脚个数信息之后便可以生成相应长度的软 JTAG 链。对应的 Verilog 文件就可以由程序生成，自动输出到 UserCell.v 文件中。具体代码如图 3-37 所示。

```
module UserCell(DRCK1, Update, Shift, SEL, TDO, reset, TDI, DataIn, DataOut);
parameter cell_num = 4;

input DRCK1, Update, Shift, SEL, TDI, reset;
output TDO;
input [cell_num-1:0] DataIn;
output [cell_num-1:0] DataOut;
reg [cell_num-1:0] ShiftReg;
reg [cell_num-1:0] LatchReg;

assign TDO = ShiftReg[cell_num-1];
assign DataOut = LatchReg;

always @(posedge DRCK1 or negedge reset)
if(~reset)
    ShiftReg <= 0;
else if(Shift)
        ShiftReg <= {ShiftReg[cell_num-2:0],TDI};
else ShiftReg <= DataIn;

always @(posedge Update or negedge reset)
if(~reset)
    LatchReg <= 0;
else if(SEL)
    LatchReg <= ShiftReg;
endmodule
```

图 3-37　构建 JTAG 软扫描链的代码

上述文件即为程序自动生成的软 JTAG 链。图 3-37 中的代码功能描述如下：

在 reset 的下降沿，软链中的所有数据复位为 0；

在 reset 为高电平、Shift 为低电平时，移位寄存器链中读入 DataIn 的数据。

在 reset 为高电平、Shift 为高电平时，移位寄存器链在 DRCK1 的上升沿从 TDI 读入数据，从 TDO 输出数据。

在 reset 为高电平、SEL 为高电平时，移位寄存器链中的数据在 Update 的上升沿被锁入锁存器，也即从 DataOut 输出。

注意到链的长度 cell_num 为 4，即为输入输出管脚数的较大者。对于不同的 Verilog 程序，只需根据输入输出管脚数修改 cell_num 即可。

最后生成顶层的 Verilog 包装文件，只需将 counter4 的输入输出接至图 3-37 中的 JTAG 扫描链，并将该扫描链连接到芯片内部的 TAP 控制器就能实现在线测试功能。例如在 Xilinx 公司的 Spartan-Ⅱ系列器件中，就有一个 BSCAN_SPARTAN2(boundary scan primitive)模块来实现 JTAG 状态机的功能，该模块的接口见 Xilinx 公司所提供的硬件库。于是将 JTAG 扫描链的相关控制信号连接至 BSCAN_SPARTAN2 模块即可，顶层代码如图 3-38 所示。

```
module top( SEL1, DRCK1, UPDATE, SHIFT, out);
output [3:0] out;
wire [3:0] TestPinIn, TestPinOut;
wire clk;
wire clr;
wire [3:0] out;
output SEL1,DRCK1,UPDATE,SHIFT;
wire DRCK1,DRCK2,RESET,SEL1,SEL2,SHIFT,TDI,UPDATE,TDO1,TDO2;

counter4 counter4_User( .clk(clk), .clr(clr), .out(TestPinOut[3:0]) );
assign {out} = TestPinOut;
assign {clk, clr} = TestPinIn;

UserCell user1 (.DRCK1(DRCK1),.Update(UPDATE), .Shift(SHIFT),
.SEL(SEL1), .TDO(TDO1),.reset(RESET),.TDI(TDI),
.DataIn(TestPinOut),      // User Cell outputs
.DataOut(TestPinIn)       // User Cell inputs
);
BSCAN_SPARTAN2 BSCAN_SPARTAN2_inst (   // Xilinx library cell
.DRCK1(DRCK1),            // Data register output - USER1 functions
.DRCK2(DRCK2),            // Data register output - USER2 functions
.RESET(RESET),            // Reset output from TAP controller
.SEL1(SEL1),              // USER1 active output
.SEL2(SEL2),              // USER2 active output
.SHIFT(SHIFT),            // SHIFT output from TAP controller
.TDI(TDI),                // TDI output from TAP controller
.UPDATE(UPDATE),          // UPDATE output from TAP controller
.TDO1(TDO1),              // Data input for USER1 function
.TDO2(TDO2)               // Data input for USER2 function
);
endmodule
```

图 3-38 在线测试 JTAG 软扫描链的顶层代码

以上基于用户电路自动生成 JTAG 软扫描链各步骤均可以由程序自动完成。该程序的流程如图 3-39 所示。在自动读入用户电路的 Verilog 代码并提取管脚信息后，先把总线信号拆分为独立的输入输出，从而确定软扫描链的长度。然后再生成输入输出名字在 JTAG 链中的位置对应表，从而由用户电路、JTAG 扫描链和 TAP 控制器生成一个顶层 Verilog 文件。该顶层文件经过编译后就能下载到 FPGA 内部，用户就可以通过电脑端口向 JTAG 软扫描链读写数据，包括向电路发送激励，读出电路的输出结果，并进行分析比较，从而达到在线调试的目的。

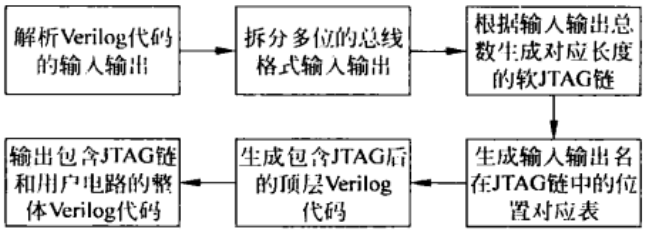


图 3-39　JTAG 软扫描链自动化设计流程图

3.7　ASIC 和 FPGA 设计规范比较

虽然专用集成电路和 FPGA 的设计流程基本相似，两者都要经过 HDL 描述的词法和语法分析、逻辑综合、工艺映射、布局布线、时序分析等过程，但是由于 FPGA 器件中所有的硬件资源都是事先制造，而不像专用芯片那样可以针对不同的需求来设计并制造硬件资源，因此两者在 HDL 的设计规范方面有所不同。早期 FPGA 被用作电路的功能原型验证时，Verilog 文件可以在 FPGA 器件上进行快速功能验证。如果功能验证通过，用户就可以直接利用这些 Verilog 文件进行专用芯片的设计。但是随着 FPGA 器件的复杂度和对专用芯片的性能和功耗要求越来越高，用户在编写 Verilog 时需要区分 ASIC 和 FPGA 不同的设计规范。

1. 深度流水线设计

在传统的通用型处理器设计中，流水线结构是提高处理器的时钟频率的主要方法。流水线结构的基本思想是把一个任务划分为数个步骤，每一个步骤只完成一个子任务。流水线结构就是使这些子任务构成顺序处理。以汽车装配为例，假设一个工人单独装配完成汽车上的一个主要部件时间 T 为 30 分钟。如果这项装配任务能够被划分为 $n=10$ 个步骤，每个步骤大概需要一个工人 2 分钟的时间，记为 t。在这里 $t<T/n$ 的原因是由于每个工人只需要熟练一个局部的子任务，而不是需要熟悉一个大任务，因此工作熟练程度可以明显提高。当然这里的前提是：一个大任务被划分成子任务时需要认真考虑，尽量把局部的工作放到一个子任务内部处理；把一些需要整个部件的翻转、移动等非局部性工作交给流水线上的机器来自动完成。流水线作业后可以安排 10 个工人分别完成所有的子任务，每两分钟就可以完成大任务的装配，在半个小时内可以装配 15 个部件。而在非流水线作业时，10 个人在半小时内只能完成 10 个部件的装配，其装配效率比流水作业差 1.5 倍。早在 1913 年福特公司就采用流水线作业的汽车装配，几乎使装配速度提高了 8 倍，实现了每隔 10 秒钟

就诞生一部汽车的神话，从而也大大降低了汽车的价格，使得更多的人能够有能力购买汽车。

数字电路设计中的同步时序电路以及 FPGA 的可编程基本逻辑单元结构设计就采用了这种流水线的思想。如第 2 章的图 2-7(a)所示，在时钟的控制下，每一块组合电路只完成一小部分的函数功能。输出的结果经过触发器或者寄存器后进行下一级流水线处理。图 2-7(a)中的示意图为两级流水处理，即它需要两个触发器。如果没有这些触发器，那么输入到输出假设需要处理的时间为 T，对应于最高时钟频率为 $f=1/T$。如果忽略触发器本身所需要的建立和保持时间(参见 3.5.5 节)，那么经两级流水线处理后，每个子组合电路就很有可能只需要 $T/2$ 的处理时间，或者说延时能够减少一半。这样电路工作频率就上升一倍。从工作频率的角度来看，只要被划分成子电路后的组合电路延时小于划分前的电路延时，那么工作频率就会提高。由于 FPGA 内部一般包含大量的时序单元触发器，即每个 LUT 输出都可以连到一个触发器，那么在电路设计时就可以进行深度流水线处理，尽量把组合电路划分成很小的单元，这样电路的工作频率就可以提高。因为对于 FPGA 器件来说，这些触发器如果不能被利用也就浪费了。同样以图 2-7(a)为例，如果我们能够找到一种电路设计方法可以进一步细分组合逻辑，使得流水级别从二级提高到四级，那么电路的工作频率就很可能进一步提高。因此，从提高 FPGA 资源利用率和电路的工作频率来说，深度流水处理都是一种非常好的处理方法。而对于专用芯片来说，虽然深度流水也能提高电路的工作频率，但是专用芯片不存在触发器被浪费的问题。因此相比较而言，基于 FPGA 的电路设计比 ASIC 对深度流水线的要求更高。

2. 锁存器和异步电路

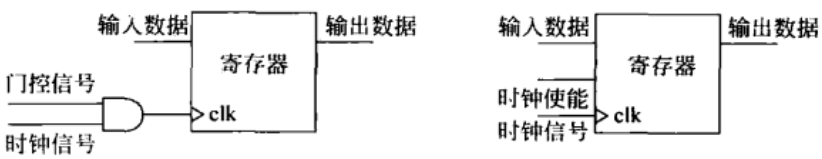
在专用芯片的性能优化方法中，锁存器和异步电路是比较常见的方法。锁存器是电平触发元件，它所需要的晶体管数目要比边沿触发器少很多。另外对于数字电路性能要求非常高的应用来说，锁存器还是经常采用的元件。由于同步电路的时钟树功耗所占比例较大，而异步电路并不需要统一的时钟，因此异步电路设计方法在低功耗专用芯片中也是比较常用的手段。但是不管是锁存器还是异步电路，它们对各元件和互连的延时要求很高。如果电路中的线网延时偏差大，那么基于锁存器的电路或异步电路很可能工作不稳定。正因为锁存器和异步电路的时序特殊性，一般的静态时序分析软件工具不能很好地支持异步电路，即不能很准确地确定电路工作的最高频率。

对于 FPGA 器件来说，电路的关键路径延时主要是互连延时，而不是可编程单元的延时。在用户设计电路时线网的互连延时并不能准确控制。互连资源延时只有在电路进行布局布线后才能完全确定。多条线网的布局布线结果是彼此关联的。如果某个区域的连接线网很稀疏，那么这些线网的互连延时就较小；反之，如果某个区域的线网分布非常拥挤，那么这些线网就会有较多的绕线，延时就会增大。有时稍微修改一下编译设置或约束条件，就可能会影响电路布局布线的结果。这就意味着对于电平触发的锁存器或者异步电路而言，有效信号的时间或电平跳变的时间和用户所要求的时序会存在比较大的偏差，很容易造成电路工作不稳定。因此对于 FPGA 的应用电路来说，一般都避免使用锁存器或异步电路设计方案。如果已有的 HDL 代码中已经使用了锁存器和异步电路，那么这些电路就应该改用触发器和同步电路的设计，以保证能够在 FPGA 上稳定工作。

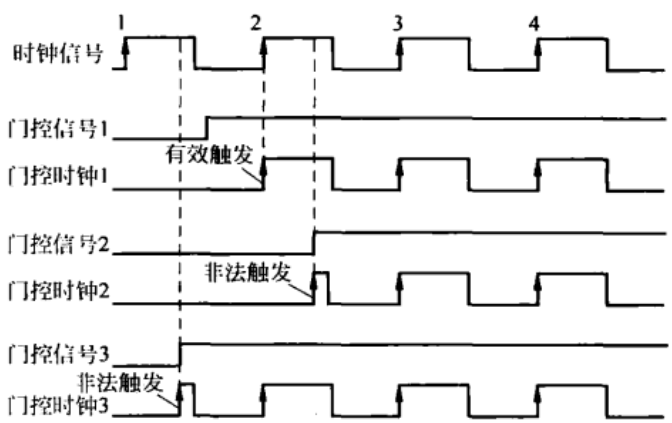
3. 门控时钟和使能时钟

在低功耗电路设计中，门控时钟技术是常用的优化方法。当电路的状态不需要改变时，就可以通过门控的方法把时钟关掉，这样可以有效地节约功耗。但是对于基于 FPGA 的电路设计来说，由于所有的硬件资源都是事先制造好的，门控信号也只能利用已有的互连资源，而不能随意改变硬件连接关系。因此，在布局布线的时候门控信号的互连延时事先不一定能够准确控制。

图 3-40(a)所示为数字电路常用的门控时钟电路，它由一个与门和一个门控信号控制寄存器的时钟端。当门控信号为高电平时，时钟信号就能经过与门正常地驱动寄存器。当门控信号为低电平时，时钟信号就被与门屏蔽。这样寄存器的时钟端就没有时钟触发输入。因此寄存器状态就保持不变。在图 3-40(c)中，“门控信号 1”的延时比较合适，因此门控后的时钟输出“门控时钟 1”所产生的上升边沿和原来时钟信号是同步的，属于有效触发。这样寄存器就能正常工作。但是如果门控信号所在的区域由于硬件资源的拥挤度较高，从而导致门控信号的延时较大，如图 3-40(c)中的“门控信号 2”所示，那么这时与门输出的“门控时钟 2”的上升边沿就不是原来时钟信号的有效边沿，而是门控信号的上升边沿。因此这时“门控时钟 2”输出的触发属于非法触发，从而导致电路寄存器很可能读取了无效的数据，并影响电路的正常工作。另一方面，如果门控信号所在的区域硬件资源的拥挤度很低，以至于门控信号的延时很小，如图 3-40(c)中的“门控信号 3”所示，那么这时与门输出的“门控时钟 3”的上升边沿也不是原来时钟信号的有效边沿，而是门控信号的上升边沿。因此这时“门控时钟 3”输出的触发信号同样属于非法触发，影响电路的正常工作。从图 3-40(c)还可以看出，门控信号的引入由于互连延时的偏差很可能会产生毛刺时钟输入，即时钟触发边沿后所



(a)门控时钟寄存器　　(b)时钟使能寄存器



(c)门控信号的不同互连延时导致的非法触发

图 3-40　门控时钟和使能时钟的比较

维持的高电平时间很短，毛刺的引入对电路工作的稳定性同样带来很大影响。

因此，在 FPGA 可编程逻辑单元内部，我们就不能像传统的设计方法一样采用门控时钟，解决办法就是采用可编程逻辑单元的“时钟使能”端。商用的 FPGA 器件中的可编程逻辑单元一般都增加了“时钟使能”输入，当“时钟使能”端有效时，外部的时钟能够正常驱动寄存器；反之，当“使能信号”无效时，外部的时钟就不能改变寄存器的状态。因此它内部的控制逻辑电路能保证寄存器的触发都是有效的，避免引入了非法触发的可能性。

4. 嵌入式 IP 核的使用

从电路设计的性能和便利性方面考虑，一般 FPGA 供应商都会提供一些 IP 库。这就像软件编译器一般都提供基本的函数库一样，例如一般 C 语言标准函数库都会提供 printf 函数。这些标准库中的函数性能一般要比用户自己编写的性能好。对于硬件电路设计来说，如果 FPGA 器件已经包含了一些粗粒度的硬件资源，例如 2.4 节讨论的嵌入式存储器、微处理器或者 DSP 模块，那么用户就没有必要再利用 FPGA 的可编程逻辑单元来实现。另外，FPGA 供应商一般都针对一些常见的应用提供了 IP 库，例如基本的通信协议、常用的数字滤波器、编解码器、FIFO 和各种存储控制器等。这些单元都是 FPGA 供应商根据器件的硬件特性事先优化过的，因此用户就没有必要自己设计 FPGA 供应商的 IP 库中已经提供的电路模块。对于 FPGA 来说，嵌入式 IP 核的种类可以划分如下。由于 FPGA 引入了可编程性，这些 IP 核的划分标准和专用芯片所使用的 IP 核会有所不同。

(1) FPGA 器件内部直接嵌入的硬核

所谓 FPGA 内部的硬核就是指把 IP 核的版图直接放到 FPGA 硬件中去。这些 IP 核在 FPGA 内部一直占据着芯片空间。如果这些 IP 核不能被用户的电路所使用，那么它们的资源就被浪费了。系统级 FPGA 器件一般都嵌入了多种粒度的硬核。图 3-41 是系统级 FPGA 内部结构示意图。其中的阵列结构除了基于 LUT 的可编程逻辑资源、可编程互连资源外，还嵌入了一些硬核，例如嵌入式微处理器核——PowerPC 或者 ARM 硬核，嵌入式 DSP 模块和嵌入式存储器核。这些硬件直接占据一定的芯片面积，并且硬核的数量在芯片制造后就完全确定了，不能事后更改。系统级 FPGA 器件还可能嵌入用于时钟管理的锁相环核、高速通信 IO 等硬核。

(2) 利用可编程逻辑单元实现的软核

FPGA 内部的软核是指利用可编程逻辑资源和可编程互连资源来实现的各种功能单元。例如图 3-41 右上方画出了一个微处理器软核，它是利用普通的 FPGA 可编程资源来实现的。如果用户电路中并没有用到微处理器，那么这些可编程资源可以被其他电路所使用。如果 FPGA 内部资源很丰富，那么在一个 FPGA 器件内部可以实现多个处理器软核。软核的硬件资源共享性是它和硬核的最基本差别。商用 FPGA 器件中一般都支持嵌入式微处理软核，例如 Nios、MicroBlaze 和 ARM 软核等。微处理器软核的一个优势是它可以根据用户的应用需求选择不同的版本，例如高性能、低功耗或者低成本等不同的实现形式。另外嵌入式微处理器软核一般还支持用户自定义指令，即用户可以根据特定的应用，把一些比较费时的计算任务利用可编程资源来实现，并把它打包成为一条用户自定义指令。这种性能是嵌入式微处理器硬核所无法提供的。

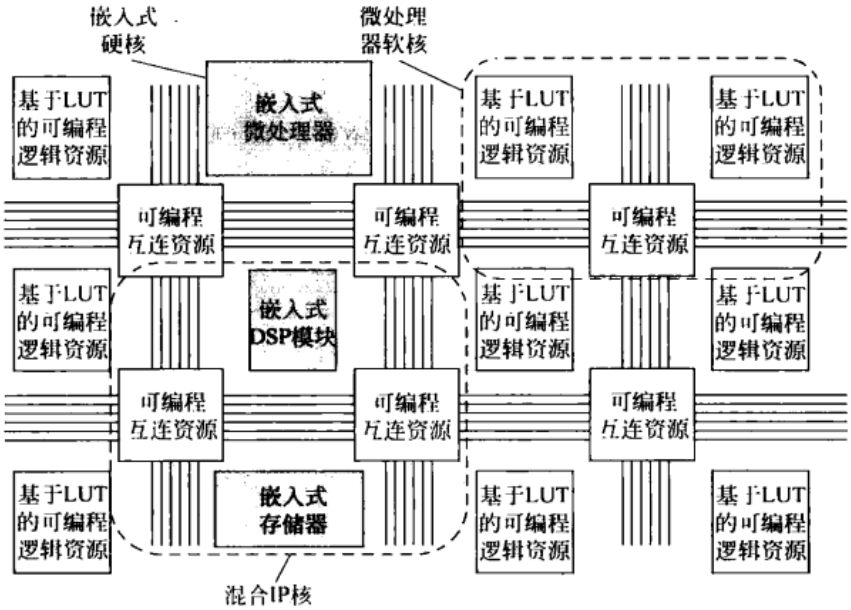


图 3-41　基于 FPGA 的各种 IP 核分类

例如乘法运算可以用 FPGA 内部的通用型可编程资源来实现，也可以利用器件内部的嵌入式 DSP 模块来实现。后者的性能肯定要比前者优越好几倍。如果 FPGA 器件内部并没有提供嵌入式 DSP 模块，那么只能用软核的方式来实现；否则如果这些嵌入式硬核不能被利用，那么它们就闲置浪费了。

（3）利用嵌入式硬核和可编程逻辑单元实现的混合 IP 核

混合 IP 是指同时利用了嵌入式硬核和 FPGA 内部通用的可编程逻辑资源和互连资源的功能模块，它有时和嵌入式软核并没有严格的区分，例如嵌入式微处理器软核会有多个版本。如果能利用嵌入式 DSP 硬核来快速实现乘法指令，那么它的性能就比较高。这时嵌入式微处理器软核也是一个软硬混合的 IP 核：它不但使用了 FPGA 的通用可编程资源，同时也利用了 FPGA 内部的嵌入式硬核。混合 IP 核是功能比较强大的 IP 核的常用方式。

综上所述，用户设计电路时首先应该尽量充分利用 FPGA 供应商所能提供的各种 IP 核，便于提高电路的性能、硬件资源利用率和便利性。在 2010 年的 ACM FPGA 国际会议论文集中，第一篇论文就介绍 Intel 公司如何把 Nehalem 处理器移植到 FPGA 器件[13]，修改代码后成为在 FPGA 可综合的处理器软核。在 RTL 代码移植过程中，就采用了门控时钟到带使能端的 FPGA 时钟、锁存器到触发器、处理器硬核等处理过程。通过只修改 5%的代码，就把整个 Nehalem 处理器移植到 5 个 FPGA 器件上运行，运行频率为 520kHz。

习题

1. 在图 3-6 中为 shiftreg_synth 电路添加“reset”输入端，使得上电后电路中各触发器被复位为 0，实现代码中的“initial”功能。

2. 下载并运行 ABC 程序，读取 shiftreg_synth 电路的 Verilog 描述文件，然后进行逻

辑优化与 LUT 映射，并显示优化后的电路结果。所用到的命令包括：read、show、fpga 等。ABC 程序的下载地址为：http://www.eecs.berkeley.edu/～alanmi/。更多的测试例子可以从 http://www1.cse.wustl.edu/～jain/cse567-08/ftp/fpga/网站下载。在 FPGA 逻辑映射中，分别把 LUT 的输入个数设置为 3、4 和 5，统计各 LUT 实际用到的输入数目，从而量化分析为何 LUT 的输入为 4 时是比较好的硬件结构。

3. 根据上一个练习中 shiftreg_synth 电路的映射结果，下载并运行 TVPack 程序和 VPR 程序，显示电路的布局布线结果。TVPack 和 VPT 工具下载地址为：http://www.eecg.toronto.edu/～vaughn/vpr/vpr.html，或者 www.eecg.utoronto.ca/vpr/。运行上个练习中的测试例子，统计互连延时在关键路径中的延时比例。

4. 从 Altera 的大学计划网站下载商用 FPGA 的结构描述 XML 文件，结合 Quartus 运行学术工具流程文档，并尝试运行 QUIP 所提供的测试电路，统计 Quartus 和学术工具的电路规模、时序性能之间的差异。QUIP 的下载地址为：http://www.altera.com/education/univ/research/quip/unv-quip.html。

5. 写出从图 3-35(a)到图 3-35(b)的中间状态图和边界扫描寄存器的值。

参考文献

[1] 丁文魁，杜淑敏. 编译原理和技术. 北京：电子工业出版社，2008

[2] Free Software Foundation 的 GCC 网上文档. http://gcc.gnu.org/onlinedocs/gccint/，2008

[3] Arthur Griffith 著. 胡恩华译. GCC 技术参考大全. 第 20 章. 北京：清华大学出版社，2004

[4] Intel 公司，http://www.intel.com/museum/archives/history_docs/index.htm#4004，2009

[5] Himanshu Bhatnagar. Advanced ASIC Chip Synthesis-Using Synopsys Design Compiler, Physical Compiler and PrimeTime. 2nd Edition, Kluwer Academic Publishers, 2002；张文俊译. 高级 ASIC 芯片综合——使用 Synopsys Design Compiler Physical Compiler 和 PrimeTime. 第 2 版. 北京：清华大学出版社，2007

[6] Samir Palnitkar. Verilog HDL——A Guide to Digital Design and Synthesis. Prentice Hall; SunSoft Press, 1996

[7] IEEE/ACM International Workshop on Logic Synthesis (IWLS) Benchmark, http://www.sigda.org/iwls/iwls2002/, 2002

[8] Daniel D Gajski. Loganath Ramachandran, Introduction to High-Level Synthesis. IEEE Design & Test of Computer, 1994, 11(4): 44-54

[9] Ian Kuon, Jonathan Rose. Measuring the Gap between FPGAs and ASICs. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2007, 26(2): 203-215

[10] Rose J, Francis R J, Lewis D, Chow P. Architecture of field-programmable gate arrays: The effect of logic block functionality on area efficiency. IEEE Journal of Solid-State Circuits, 1990, 25(5): 1217-1225

[11] Betz U, Rose J, Miurquardt A. 著. 王伶俐，杨萌，周学功译. 深亚微米 FPGA 结构与 CAD 设计. 北京：电子工业出版社，2008

[12] Sachin Sapatnekar. Timing. Boston, Kluwer Academic Publishers, 2004

[13] Graham Schelle, Jamison Collins, Ethan Schuchman, et al. Intel nehalem processor core made FPGA synthesizable. ACM 18th International Symposium on Field-Programmable Gate Array (FPGA'2010), Monterey, CA, 2010: 3-12

第 4 章

基于 FPGA 的嵌入式系统硬件设计

4.1 嵌入式系统及其 FPGA 实现

嵌入式系统(embedded system)是一种嵌入在受控设备内部,为特定应用而设计的专用计算机系统。与个人计算机这样的通用计算机系统不同,嵌入式系统执行的是带有特定要求且预先明确定义的任务。由于嵌入式系统只针对一项专用的任务,设计人员能够对它进行性能优化,减小尺寸并降低成本。因为嵌入式系统通常进行大量生产,所以单个系统的性能、尺寸和成本优化就显得非常重要。

嵌入式系统已广泛融入现代生活当中。例如,消费类电子产品中的手机、数字相机和摄像机、个人数字助理等;家用电器中的微波炉、洗衣机、DVD 播放器等;办公设备中的打印机、复印机、扫描仪等;网络设备中的交换机和路由器等。

虽然嵌入式系统应用领域很广,但具有以下共同的特性:

(1) 专用的功能:一个嵌入式系统通常重复地执行一个特定的程序。有时由于尺寸的限制,数个程序在一个系统中被调进或调出,它运行的程序也可升级到新的版本,但是一个嵌入式系统往往只针对固定的应用实现专用的功能。

(2) 严格的约束条件:所有的计算系统在设计时都有许多约束条件和各种需求。但是嵌入式系统的约束条件特别严格,诸如成本、尺寸、速度和功耗等,这些限制条件均用于评定系统实现的总体性能。

(3) 实时的反应:嵌入式系统必须能够连续地对系统环境的各种变化做出可靠的反应,并且要实时计算出确定的结果。

(4) 软硬件混合的设计技术:一般嵌入式系统由嵌入式处理器、系统的外围硬件设备、嵌入式操作系统和系统应用软件等四部分组成。它是以微处理器为核心的数字系统,涉及到硬件和软件两方面的设计技术。嵌入式系统的处理器可以是通用微处理器、微控制器或 DSP 处理器等。

4.1.1 FPGA 在嵌入式系统中的应用

嵌入式系统的应用领域不断地扩展,然而嵌入式系统的设计却面临着严峻的挑战。嵌入式系统的设计必须实现全部的预期功能,同时又要针对大量的设计指标进行优化。除去一次性工程(NRE)成本外,对设计系统的技术要求还包括单元成本、尺寸、性能、功耗、灵活

性、样机时间、上市时间、可维护性、故障率和安全性等。

例如，基于纳米级半导体工艺技术所设计的 ASIC 在技术上就非常具有挑战性，而且成本和风险都很高。如图 1-13 所示，2006 年期间 65nm 集成电路的掩膜成本就高达数百万美元，加上整个研发团队所需要的设计与验证成本，纳米级 EDA 工具版权和最终的产品测试总共需要数千万美元的成本。如果在流片后又发现新的问题，那么必须重新流片，又要支付数百万美元的掩膜费用。但是对于利用 FPGA 的嵌入式系统设计，就可以在很大程度上节省流片费、昂贵的 EDA 工具版权和芯片本身的测试成本。这是因为 FPGA 供应商已经对芯片作了严格的测试，用户并不需要自己流片；并且 FPGA 的设计软件工具一般比专用芯片的 EDA 工具便宜很多，甚至在购买芯片时就可以免费授权。另外，基于 FPGA 的系统设计非常灵活，可以根据需求变换和及时修改更新。

从系统对设计成本、上市时间以及灵活性的需求等方面考虑，以 FPGA 来实现可配置的嵌入式系统已越来越广泛。随着半导体工艺技术的进步和软件工具链的进一步完善，FPGA 器件能够进一步降低成本，减少功耗并提高性能，使得 FPGA 成为中小批量生产的应用器件。它的应用范围从早期的原型验证、通信系统等应用扩展到低成本消费电子类等产品。目前先进的 FPGA 器件包含数十万个逻辑单元、嵌入式处理器和 DSP 模块，以及高速 IO 模块，并配有嵌入式系统开发工具，从而很好地满足了嵌入式系统设计的需要。可以相信，基于 FPGA 的嵌入式系统实现将会更加普遍。

4.1.2 FPGA 在可编程片上系统设计中的应用

随着半导体工艺的迅速发展，以及 VLSI 设计的普及化，在单个芯片上已能够集成一个功能完整的复杂系统，这就是片上系统(SoC)。SoC 通常由一个主控单元和一组功能模块构成。主控单元一般是一个微处理器核，也可以是一个 DSP 核。在这个主控单元周围，根据系统所需的功能配置系统的功能模块，完成信号的接收、预处理、转换和输出等任务。片上系统技术通常应用于小型的、日益复杂的电子设备。例如，声音检测设备的片上系统是在单个芯片上为所有用户提供包括音频接收端、模数转换器(ADC)、微处理器、必要的存储器以及输入输出逻辑控制等设备。

可编程片上系统(system on a programmable chip，SOPC)是一种基于 FPGA 的灵活、高效的片上系统解决方案，它是以嵌入式微处理器和硬件可编程性为核心的嵌入式系统[1]。首先，它是一个片上系统，即由单个芯片完成整个系统的主要功能。其次，它是可编程系统，具有灵活的设计方式，可裁减、可扩充、可升级，并具备软硬件在线系统可编程的功能。可编程器件内具有小容量高速 RAM 资源，和丰富的 IP 核资源可供灵活选择使用；可编程器件内还具有足够的可编程逻辑资源，用以灵活地实现各类外围硬件电路。处理器的调试接口和 FPGA 编程接口共用或并存。有些可编程器件内还可能包含部分可编程模拟电路。整个系统采用单芯片设计与封装，功耗很低。因此，SOPC 系统会降低成本，提高系统整体性能，缩短设计迭代周期，降低硬件系统设计风险，延长了系统的生命周期。如图 4-1 所示，Gartner Dataquest 市场调查公司提供的数据表明，从 2000 年起在基于 FPGA 的电子产品设计中，包含微处理器的 SOPC 设计所占的比例稳步增长，并将逐步成为基于 FPGA 的嵌入式系统设计的主流技术。

SOPC 设计涵盖了嵌入式系统设计技术的全部内容，除了以处理器和实时多任务操作系统(run-time operating system，RTOS)为中心的软件设计技术、以印刷电路板和信号完整

性分析为基础的高速电路设计技术以外，SOPC 还涉及软硬件协同设计技术。两大 FPGA 厂商 Altera 和 Xilinx 都提供了设计工具和包括微处理器在内的大量 IP 核用于支持可编程片上系统的开发。

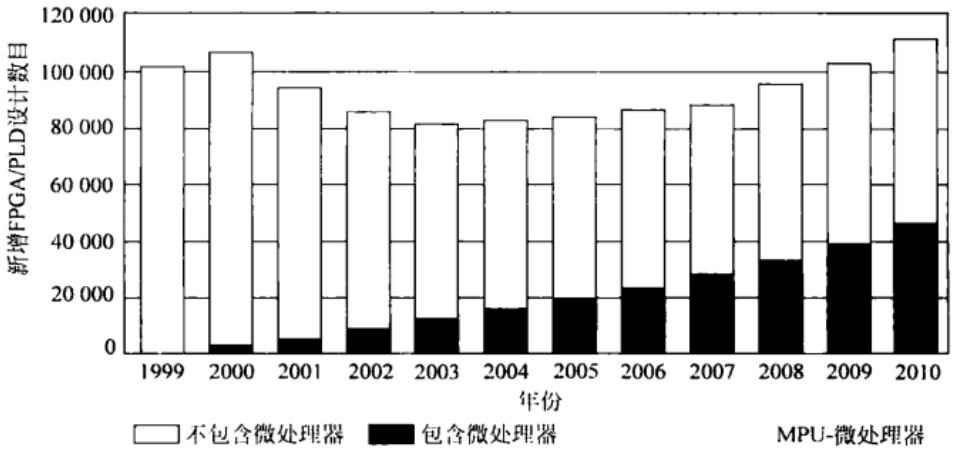


图 4-1　基于 FPGA 的产品设计增长趋势[2]

4.2　嵌入式微处理器

嵌入式微处理器(microprocessor unit，MPU)由通用计算机中的 CPU 演变而来。与通用计算机中的 CPU 不同的是，在嵌入式系统应用中，微处理器通常只保留和嵌入式应用紧密相关的功能部件，去除其他的不需要的部件，这样就以最低的功耗和资源实现嵌入式应用的特定要求。嵌入式微处理器在软件配置上常常可以运行嵌入式操作系统，从而能够适应比较高端的嵌入式系统需求。

嵌入式微处理器种类很多，大多采用精简指令集(RISC)结构，典型的如 ARM、PowerPC、MIPS 等。与 Intel x86 架构统治通用微型计算机市场的状况不同，在嵌入式系统领域还没有一种 MCU 占有绝对的优势地位。

如 2.4.2 节所述，在 SOPC 设计中还可以利用 FPGA 的可编程逻辑资源搭建微处理器软核。当强调它是一个 IP 核时，我们称之为微处理器软核；当强调它是一个处理器时，我们称之为软核处理器。有时我们不会区分这两者在名称上的细微差别。与硬核微处理器不同，软核微处理器在设计时可以进行裁减，以达到处理器性能与可编程逻辑资源占用的平衡。为满足嵌入式系统设计中的实时性要求，还可以通过自定义指令或协处理器接口，利用额外的可编程逻辑资源对软核处理器进行扩展，把一个复杂的标准指令序列简化为一条用硬件实现的单个指令，以实现算法的加速。此外，在大容量的 FPGA 中可以方便地放置多个软核处理器，实现多核并行计算。

以下分别介绍在 SOPC 设计中常用的几种嵌入式微处理器。

4.2.1　ARM

ARM 处理器是一种 32 位 RISC 微处理器架构，是目前应用最为广泛的嵌入式微处理器。ARM 处理器的发展路线刚好和 Intel 公司的奔腾处理器完全相反。后者的芯片面积、

功耗、性能、核数不断增长，而 ARM 处理器充分利用了最新的半导体工艺技术，在不断提高性能的前提下，逐步降低处理器的面积、功耗。由于低功耗等优点，ARM 处理器非常适用于移动通信领域和各种消费类电子产品，从可携式装置（移动电话、多媒体播放器、掌上型电子游戏和计算器）到电脑外设（硬盘、桌上型路由器）等。目前大约 98%的手机使用了 ARM 架构的处理器。Altera 公司的 Excalibur 系列 FPGA 中就嵌入了 ARM 处理器硬核。

4.2.2 PowerPC

PowerPC 是 20 世纪 90 年代 Motorola 和 IBM 联合为 Apple 公司的 MAC 微型计算机开发的 CPU 芯片。PowerPC 架构的特点是可伸缩性好、方便灵活。PowerPC 处理器具有多种系列的产品形式，支持从高性能服务器到嵌入式系统等应用领域。其中 PowerPC 4xx 是 IBM 公司偏向嵌入式系统开发的微处理器系列，具有优异的性能、较低的功耗和散热等特点。Xilinx 公司的高端 FPGA 芯片中嵌入了 PowerPC 4xx 处理器硬核。

4.2.3 Nios Ⅱ

Nios Ⅱ 软核处理器是 Altera 公司推出的第二代 32 位 RISC 嵌入式处理器 IP 核，是一款用户可配置的软核处理器。Nios Ⅱ 处理器包括 3 种配置形式：Nios Ⅱ/f（fast，快速型）——最高的系统性能，中等 FPGA 使用量；Nios Ⅱ/s（standard，标准型）——高性能，低 FPGA 使用量；Nios Ⅱ/e（economy，经济型）——低性能，最低的 FPGA 使用量。如图 4-2 所示，这 3 种实现形式具有 32 位处理器的基本结构单元——32 位指令位宽，32 位数据和地址路径，32 位通用寄存器和 32 个外部中断源；使用同样的指令集架构，100%二进制代码兼

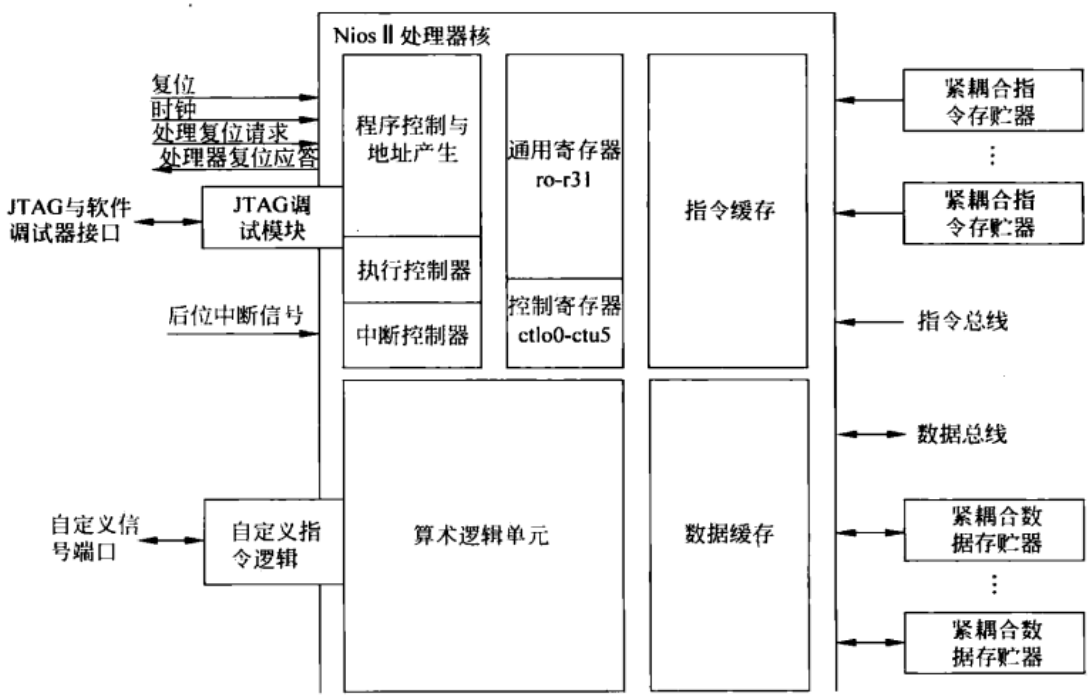


图 4-2 Nios Ⅱ处理器结构框图[3]

容。设计者可以根据系统需求的变化更改处理器实现形式，选择满足性能和成本的最佳方案，而不需修改已有的代码。

Nios Ⅱ 软核处理器是针对 Altera 公司 FPGA 器件结构进行优化的免费 IP 核，在 FPGA 器件中所占的硬件资源相对较少。其最大的特点在于可配置性，即用户可以根据自己的标准定制处理器，按照需要选择合适的外设、存储器和接口，此外还可以轻松地集成一些专用的功能，如定制指令集和定制用户逻辑模块等。

4.2.4　MicroBlaze 和 PicoBlaze

MicroBlaze 是基于 Xilinx 公司 FPGA 的软核微处理器。它具有运行速度快、占用资源少、可配置性强等优点。和其他外设 IP 核一起，可以完成可编程片上系统的设计。MicroBlaze 处理器采用 RISC 架构和哈佛结构的 32 位指令和数据总线，并具有协处理器接口。

MicroBlaze 采用 CoreConnect 总线接口，与 PowerPC 相同。当系统性能不能满足设计要求时，用户可以方便地将基于 MicroBlaze 软核处理器的系统升级为采用 PowerPC 硬核微处理器的系统，不需要改变外设电路的设计。

PicoBlaze 是 Xilinx 公司提供的 8 位微处理器软核。它由 VHDL 语言设计，不需要预编译，可直接由布局布线工具嵌入到 FPGA 器件中。PicoBlaze 只占用非常少的可编程资源，在 SpartanIIE FPGA 中只需占用 76 个 Slice，不到 XC2S300E 器件 2%的资源。因此在一个系统设计中可以包括多个 PicoBlaze 软核，而不必担心其资源的占用。PicoBlaze 适用于对性能要求不太高，但是控制逻辑较为复杂的数字系统，可以简化系统设计。

4.3　片上总线

在 SoC 的设计过程中，最具特色的是 IP 复用技术，即用户选择所需功能的 IP 核，然后集成到一个芯片。由于 IP 核的设计千差万别，IP 核的接口就成为构造 SoC 的关键。片上总线(on-chip bus，OCB)是实现 SoC 中 IP 核连接最常见的技术手段，它以总线方式实现 IP 核之间的数据通信。与板上总线不同，片上总线不用驱动底板上的信号和连接器，使用更简单，速度更快。片上总线规范一般需要定义各个模块之间初始化、仲裁、请求传输、响应、发送接收等过程中的驱动、时序、策略等关系。

片上总线与板上总线应用范围不同，存在着较大的差异。其主要特点如下：

(1) 片上总线要尽可能简单。首先结构要简单，这样可以占用较少的逻辑单元；其次时序要简单，以利于提高总线的速度；第三接口要简单，如此可减少与 IP 核连接的复杂度。

(2) 片上总线要有较大的灵活性。由于片上系统应用广泛，不同应用对总线的要求各异，因此片上总线需要具有较大的灵活性，主要表现为：其一，多数片上总线的数据和地址宽度都可变，如 AMBA AHB 支持 32～128 位数据总线宽度；其二，部分片上总线的互连结构可变，如 Wishbone 总线支持点到点、数据流、共享总线和交叉开关四种互连方式；其三，部分片上总线的仲裁机制灵活可变，如 Wishbone 总线的仲裁机制可以完全由用户定制。

(3) 片上总线对功耗的要求严格。这就要求在实际应用时，总线上各种信号尽量保持不变，这样就能节约动态功耗。另外也要求多采用单向信号线，从而进一步降低功耗，同时也简化了时序。片上总线的输入数据线和输出数据线是分开的，也没有板上总线常有的地

址线与数据线复用的现象。

片上总线一般分为高性能的系统总线(system bus)与低功耗的外围总线(peripheral bus)两个部分。系统总线用来连接微处理器、DMA 控制器、片上存储器和其他具有高带宽通信要求的设备。系统总线可以连接多个主设备,因而需要总线仲裁来控制各个主设备对总线的访问请求。系统总线的特点是高速、高带宽。外围总线用来连接对速度要求不高的各类外围设备。在外围总线上通常只有一个主设备,因此其协议比系统总线协议简单。外围总线的特点是低速、低带宽,但是它要满足低功耗、重用性等方面的要求。SoC 设计中利用总线的分层技术可以使各种不同特性的模块与总线更好地连接,提高总线的运行效率。

片上总线尚处于发展阶段,不像微机总线那样成熟,目前还没有统一的标准。因此各大厂商和组织纷纷推出自己的标准,以便在未来的 SoC 片上总线标准中占有一席之地。其中比较有影响的片上总线有 AMBA 系列总线、IBM 的 CoreConnect 总线、OCP(open-core protocol)总线、Wishbone 总线、Avalon 总线等。

4.3.1 Avalon 总线

Avalon 总线架构是由 Altera 公司开发的片上总线架构。总的来说,Avalon 是一种相对简单的总线架构,主要用来将处理器和外围设备集成到片上可编程系统中,并规定了主设备和从设备的端口连接方式以及时序关系。Avalon 总线架构的基本设计目标如下:

(1) 简洁性:提供一种易于理解的协议。

(2) 低成本:为总线逻辑提供优化的资源,从而节约可编程逻辑资源。

(3) 同步性:基于同步操作,易于与片上的其他用户逻辑集成,避免了复杂的时序约束和分析过程。

Avalon 总线拥有多种传输模式,以适应不同设备的要求。Avalon 总线的基本传输模式是在主设备和从设备之间传输一个字。一次传输过后,总线可以立刻进行下一次传输,而且与上一次传输的目的设备和源设备无关。Avalon 总线还支持流水线传输(pipelined transfer)和突发传输(burst transfer)等模式。这些传输模式使得在一次总线传输中,在设备之间能够完成多个数据单位的交换。

Avalon 总线支持多个总线主设备,允许单个总线事务在设备之间传输多个数据单元。这一多主设备结构为构建 SOPC 系统提供了极大的灵活性,并且能适应高带宽的设备。例如,一个主设备可以进行直接存储器访问(DMA)传输,从设备到存储器传输数据时不需要处理器干预。

Avalon 总线主设备和从设备的交互采用了"从端仲裁"技术,在多个主设备试图访问同一个从设备时,用于决定哪个主设备获得访问权。这使其具有以下两个优点:

(1) 仲裁的细节被封装到 Avalon 总线内,主设备和从设备的接口与总线上设备数目无关。

(2) 多个主设备能够同时执行总线传输,只要它们不在同一时钟周期访问同一个从设备。

另外 Avalon 总线是为 SOPC 环境而设计的,整个总线的互连电路都由 FPGA 内部的逻辑单元实现。Avalon 总线具有以下基本特点:

(1) 所有设备的接口与 Avalon 总线时钟同步,不需要复杂的握手和应答机制,这样就简化了 Avalon 总线的时序行为,而且便于集成高速设备。Avalon 总线以及整个系统的性

能可以采用标准的同步时序分析技术来评估。

(2) 所有的信号都是高电平或低电平有效，便于信号在总线中高速传输。在 Avalon 总线中，由数据选择器代替三态缓冲器来决定哪个信号驱动哪个设备。因此外设即使在未被选中时也不需要将输出置为高阻态。

(3) 为了方便外设的设计，地址、数据和控制信号使用分离的、专用的端口。外设不需要识别地址总线周期和数据总线周期，也不需要在未被选中时使输出无效。分离的地址、数据和控制通道还简化了与片上用户自定义逻辑的连接。

图 4-3 展示了 Avalon 总线架构的一个例子。

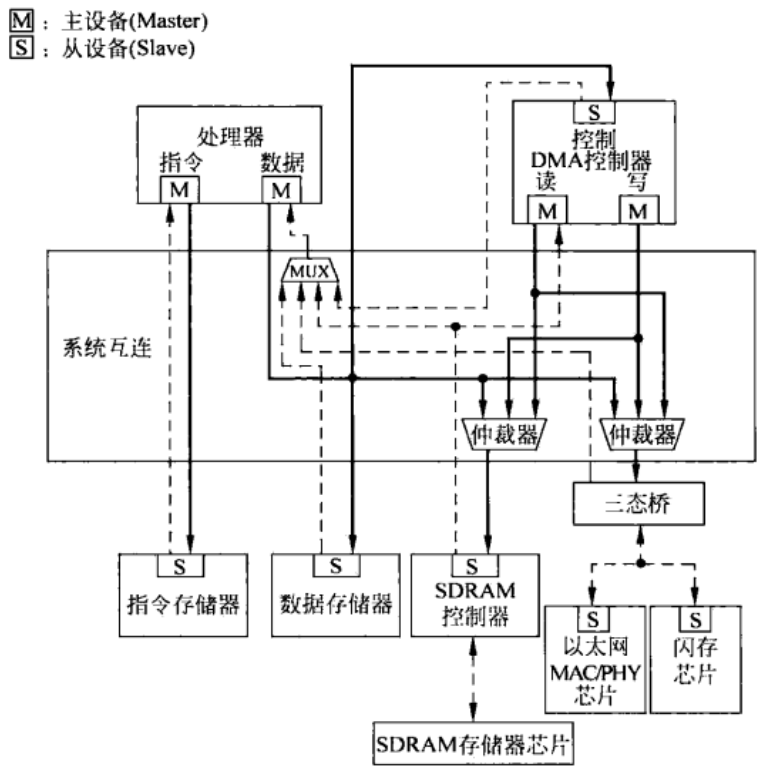


图 4-3　Avalon 总线架构[4]

片上总线的接口信号大部分都是可选的，即一个 IP 核不需要定义全部的接口信号，只需定义它的数据传输方式所要求的少量信号即可。表 4-1 示出了一个典型的 Avalon 从设备接口所包含的信号。

表 4-1　常用 Avalon 从设备接口信号

信号类型	宽度	方向	说明
clk	1	In	时钟
reset	1	In	复位
chipselect	1	In	片选
address	1～32	In	地址
read	1	In	读请求

续表

信号类型	宽度	方向	说明
readdata	1～32	Out	读数据
write	1	In	写请求
writedata	1～32	In	写数据
irq	1	Out	中断请求

图 4-4 显示了典型的 Avalon 从设备接口信号的时序。

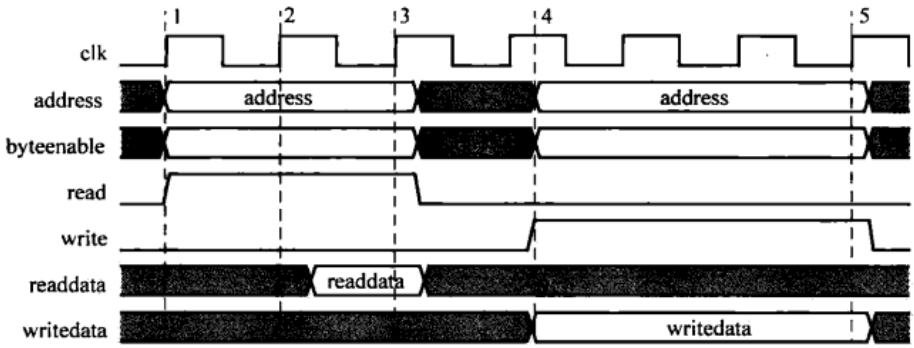


图 4-4 Avalon 从设备基本读写时序[5]

4.3.2 AMBA 总线

先进微控制器总线架构(advanced microcontroller bus architecture,AMBA)是由 ARM 公司于 1995 年开发的总线架构标准。由于 ARM 公司在业界的影响以及 AMBA 总线架构在业界 SoC 系统产品,特别是基于 ARM 处理器内核系统芯片的大量实际应用,使得 AMBA 总线架构逐渐成为 SoC 系统设计业界采用的主要片上总线架构。

AMBA 总线定义了片上系统内部的通信标准,以设计出高性能的嵌入式微处理器。在 AMBA 总线规范中定义了三类不同的总线。

(1) Advanced High-performance Bus(AHB)

AHB 总线是一个高性能、高时钟频率的系统总线。AHB 作为高性能系统的中枢总线,它支持处理器、片上设备以及片外存储接口与低功耗外设的有效连接。同时,AHB 的设计使得基于系统集成和自动测试技术的设计流程更加方便有效。

(2) Advanced System Bus(ASB)

ASB 是 AMBA 规范早期版本定义的另一种系统总线,同样支持处理器、片上设备以及片外存储接口与低功耗外设的有效连接。

(3) Advanced Peripheral Bus(APB)

APB 总线是为低功耗的外围设备设计的。APB 为支持外围设备而进行了降低功耗以及接口复杂性的优化设计,并且 APB 通过连接桥可以与任何种类的高速系统总线实现连接。

由 AMBA 构成的片上系统如图 4-5 所示,通常可以归纳为以下几个部分:

(1) 总线主设备:任何一个功能 IP 核只要能提供 AMBA 高速总线所需要的主设备接

口就可以成为 AMBA 总线架构中的主设备。

(2) 总线从设备：相应的总线也存在从设备，任何一个功能 IP 核也可以成为 AMBA 总线架构中的从设备。同样，IP 核也需提供 AMBA 总线架构所需的从设备接口，包括高速总线 AHB、ASB，低速外设总线 APB。

(3) 总线互连模块：这是整个 AMBA 总线架构的核心部分。它负责完成数据、地址和控制信号的选择传输以及仲裁主设备分时占有总线的任务。AMBA 总线架构采用了非常灵活的两级总线架构，有适应于高带宽设备的高速总线 AHB 和 ASB，也有适应于低带宽、低功耗的外设总线 APB，高速总线与低速总线之间通过连接桥模块实现互连。

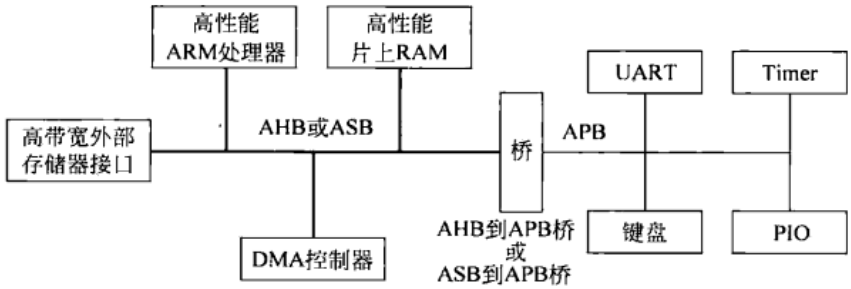


图 4-5　AMBA 总线结构[6]

4.3.3　CoreConnect 总线

CoreConnect 总线则是由 IBM 公司提出的片上总线架构。该总线架构定义了三类总线：处理器局部总线(processor local bus，PLB)、片上外围总线 (on-chip peripheral bus，OPB)和设备控制寄存器总线(device control register，DCR)。PLB 总线一般用于连接高带宽、高速度的 IP 设备，类似于 AMBA 总线架构当中的 AHB 总线；而 OCB 总线则连接低带宽的 IP 设备，类似于 AMBA 总线架构中的 APB 总线，不同的是 OPB 允许有多个主设备存在，而 APB 规定只有一个主设备。PLB 和 OPB 总线都拥有自己的仲裁模块。PLB 支持四级深度读流水线，两级深度写流水线，同 AMBA 一样也支持成组传输、分离传输。它采用分离读写总线。DCR 总线用于在 CPU 的通用寄存器和外设的控制寄存器之间传递数据。采用独立的 DCR 总线消除了外设控制寄存器在内存地址空间中的映射，减少了读取操作，提高了 PLB 的带宽。

图 4-6 为采用 CoreConnect 总线的片上系统架构。

4.3.4　Wishbone 总线

Wishbone 是由 Silicore 公司制定的片上总线协议。它是一种完全开放的适用于 SoC 内部 IP 模块互连的总线协议，用户可免费使用并且还可以对其进行修改。Wishbone 总线结构极其简单、灵活，又是公开免费的，因而得到了众多支持。

Wishbone 总线采用一种单一的总线方式支持多主设备和多从设备，可配 8～64 位的数据宽度和 64 位的地址宽度，支持单字和块传输模式，具备 RMW(read-modify-write)指令周期、多时钟同步共享总线结构、握手信号传输方式和集中仲裁工作方式，并且还提供了用户扩展功能。Wishbone 协议不仅考虑了高速设备之间所需要的高速数据通道，而且提供了握

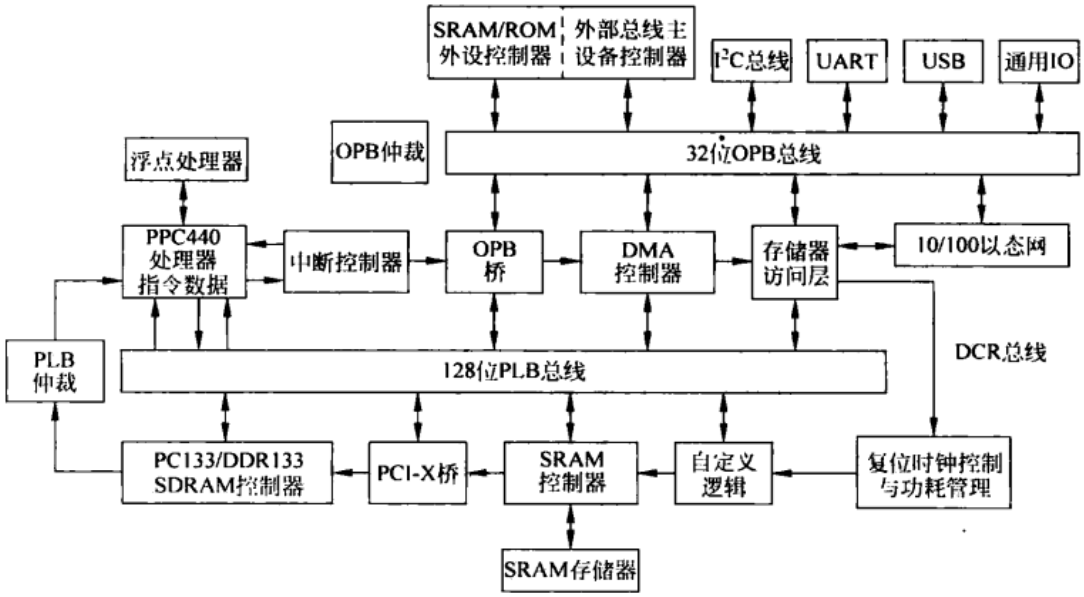


图 4-6 CoreConnect 总线架构

手信号以兼顾慢速设备的数据传递，非常适合多个异速总线之间的相互连接。

Wishbone 总线允许 IP 之间的连接可以采用多种方式：点对点(point-to-point)、数据流(data flow)、共享总线(shared bus)和交叉开关(crossbar switch)。点对点方式适合于将两个 IP 核直接连接在一起；数据流方式适合于数据按顺序进行处理的应用场合；共享总线方式则同 AMBA、CoreConnect 等总线结构相近；交叉开关方式能够让多个主设备同时访问不同的从设备。

4.3.5 四种片上总线的比较

尽管片上总线的标准很多，但这些总线标准间的特征差别并不大，所采用的结构和传输亦基本类似。这里对上述四种片上总线结构作一个总结比较，见表 4-2。

表 4-2 四种片上总线的比较

特 征	Avalon	AMBA	CoreConnect	Wishbone
层次化	/	Y	Y	/
单/双向总线	单	单	单	单
连接方式	交叉开关	共享	共享	多种方式
流水线传输	Y	Y	Y	/
集中/分布仲裁	分布	集中	集中	集中

四种总线都采用了单向数据项传输；AMBA 和 CoreConnect 总线提供了分层的总线协议，而 Avalon 和 Wishbone 总线采用单一的总线协议，但 Wishbone 总线也可配置成分层的总线结构；AMBA 和 CoreConnect 采用共享总线，Avalon 采用交叉开关，Wishbone 总线不规定统一的连接方式，可由用户灵活配置；Wishbone 总线协议相对比较简单，对流水线等高级传输方式支持比较弱一些。

4.4 自定义外设电路的设计

可复用的IP核是SoC设计的基础。IP核是预先设计、经过验证和优化的硬件模块，SoC由多个IP核互相连接而成，SoC的设计因而包括IP核的设计与系统集成两个阶段。在嵌入式系统中，通常将直接或间接地通过总线与微处理器相连，并提供存储、输入或输出等功能的硬件模块或设备称为外围设备，简称外设。在SOPC设计中，大多数IP核都属于外设。

各类SOPC设计工具的组件库已提供了包含微处理器、内存接口、总线桥以及常用外设等IP组件。然而在实际的系统设计中，这些组件往往不能满足特定的需要，还需加入用户自定义的外设。

在SOPC设计工具中一般都提供了通用输入输出(GPIO)组件，通过一个或多个GPIO组件可以连接任意的硬件模块。在传统的嵌入式系统设计中，如果某个外设的接口和系统的总线接口不一致，只能通过GPIO组件间接地连接到总线上。然而在系统设计中过多地采用这样的连接外设会对软件开发带来很大的负担。这种设计方法需要较多的软件代码与用户通信逻辑，软件的可读性也比较差。在SOPC设计中，利用FPGA的可编程特性，便能在用户自定义外设中实现总线接口，直接与总线相连。它与采用GPIO的方式相比，减少了软件的工作量，增加了系统的可理解性与可维护性。更进一步来说，对于一个经常使用的外设，还可以将它加入到组件库中，提高了外设的重用性。

4.4.1 自定义外设的结构

为提高自定义外设的可重用性，在结构上通常将自定义外设分为两个功能模块：提供了外设基本功能的任务逻辑电路，以及为外设的数据输入输出和对外设的控制提供标准的接口电路，它通常直接与总线相连。任务逻辑的设计取决于自定义外设要实现的功能。下面主要介绍接口电路的设计。

外设通过总线与处理器或其他外设通信。硬件模块间的直接相连可以自由定义数据线与控制线等接口信号，但总线连接与此不同。总线的接口信号是由总线协议规定的，特别是数据线的宽度是事先固定的，然而外设对外通信可能需要更多的接口信号，解决这一问题的常规手段是使用设备寄存器。

外设通常需要定义多个寄存器，例如数据输入寄存器、数据输出寄存器、状态寄存器、控制寄存器等。每个寄存器的位数不超过总线数据线的宽度。处理器可通过地址线的选择读写不同的寄存器，实现对外设的数据通信与控制。

外设往往还需要主动通知处理器某些事件的发生，这一般是通过中断信号实现的。中断信号分为电平中断和边沿中断两种方式。电平中断通过将中断信号置为高电平或低电平来通知处理器，处理器响应中断后需主动清除中断。边沿中断通过中断信号的上升沿或下降沿来通知处理器，外设通常产生一个脉冲信号，不需要处理器清除中断。外设具体使用哪一种中断方式取决于系统中使用的处理器类型，许多微处理器同时支持两种中断方式。

除了上述功能外，外设接口电路还可以包含输入/输出FIFO(first input first output)模块，用于实现处理器和外设间的速度匹配，以及DMA电路实现外设间的直接数据传输。

图 4-7 是一个外设接口电路的结构框图，显示了完整的接口功能。

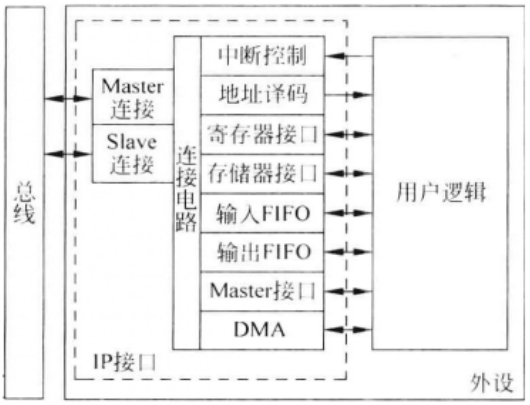


图 4-7　外设接口电路结构框图

4.4.2　基于 Xilinx FPGA 的外设接口设计实例

Xilinx 提供了“外设创建向导”帮助用户创建自定义外设。外设创建向导生成两个 VHDL 文件，分别包含总线接口和任务逻辑的代码框架。其中总线接口文件使用了预先设计的 IPIF(IP interface)模块。完整的 IPIF 模块为自定义外设提供了基于 PLB 或 OPB 总线主设备与从设备总线访问接口、用户寄存器或用户存储器访问接口、读/写 FIFO、中断与复位逻辑以及 DMA 等功能，如图 4-8 所示。

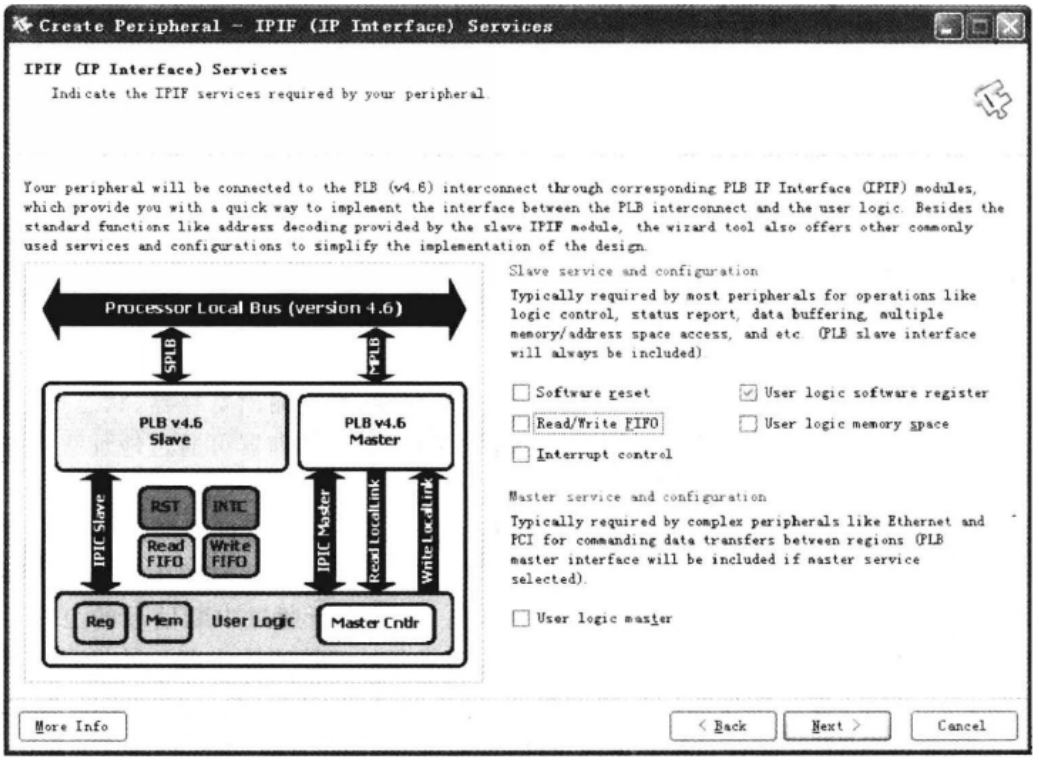


图 4-8　Xilinx 外设创建向导

图 4-9 所示为向导生成的具有两个寄存器的任务逻辑代码框架，即 user_logic.vhd 的片断代码。其主要内容是设备寄存器的读写操作，以及产生总线应答信号的代码。用户可以在此基础上添加自己的功能代码，也可以修改或简化生成的代码。用户不必修改总线接口文件的内容。

```
entity user_logic is -- 用户自定义任务逻辑模块
  port
  (
    -- 在此添加自定义端口
    -- 总线协议端口
    Bus2IP_Clk   : in  std_logic;          -- 时钟
    Bus2IP_Reset : in  std_logic;          -- 复位
    -- 数据输入
    Bus2IP_Data  : in  std_logic_vector(0 to C_SLV_DWIDTH - 1);
    -- 字节选择
    Bus2IP_BE    : in  std_logic_vector(0 to C_SLV_DWIDTH/8 - 1);
    -- 寄存器读使能
    Bus2IP_RdCE  : in  std_logic_vector(0 to C_NUM_REG - 1);
    -- 寄存器写使能
    Bus2IP_WrCE  : in  std_logic_vector(0 to C_NUM_REG - 1);
    -- 数据输出
    IP2Bus_Data  : out std_logic_vector(0 to C_SLV_DWIDTH - 1);
    IP2Bus_RdAck : out std_logic;          -- 读应答
    IP2Bus_WrAck : out std_logic;          -- 写应答
    IP2Bus_Error : out std_logic           -- 错误
  );
end entity user_logic;

architecture IMP of user_logic is
  -- 在此添加自定义信号
  ---------------------------------------------
  -- 寄存器及其读写信号
  signal slv_reg0           : std_logic_vector(0 to C_SLV_DWIDTH - 1);
  signal slv_reg1           : std_logic_vector(0 to C_SLV_DWIDTH - 1);
  signal slv_reg_write_sel : std_logic_vector(0 to 2);
  signal slv_reg_read_sel   : std_logic_vector(0 to 2);
  signal slv_ip2bus_data    : std_logic_vector(0 to C_SLV_DWIDTH - 1);
  signal slv_read_ack       : std_logic;
  signal slv_write_ack      : std_logic;

begin
  -- 在此添加自定义逻辑电路

  slv_reg_write_sel <= Bus2IP_WrCE(0 to 2);
  slv_reg_read_sel  <= Bus2IP_RdCE(0 to 2);
  slv_write_ack      <= Bus2IP_WrCE(0) or Bus2IP_WrCE(1);
  slv_read_ack       <= Bus2IP_RdCE(0) or Bus2IP_RdCE(1);

  SLAVE_REG_WRITE_PROC : process( Bus2IP_Clk ) is -- 寄存器写
  begin
```

图 4-9　具有两个寄存器的任务逻辑代码框架

```
    if Bus2IP_Clk'event and Bus2IP_Clk = '1' then
      if Bus2IP_Reset = '1' then
        slv_reg0 <= (others => '0');
        slv_reg1 <= (others => '0');
      else
        case slv_reg_write_sel is
          when "10" =>    -- 寄存器 0
            for byte_index in 0 to (C_SLV_DWIDTH/8) - 1 loop
              if ( Bus2IP_BE(byte_index) = '1' ) then
                slv_reg0(byte_index * 8 to byte_index * 8 + 7) <=
                    Bus2IP_Data(byte_index * 8 to byte_index * 8 + 7);
              end if;
            end loop;
          when "01" =>                    -- 寄存器
            ...
         when others => null;
        end case;
      end if;
    end if;
  end process SLAVE_REG_WRITE_PROC;

  SLAVE_REG_READ_PROC : process( slv_reg_read_sel,
              slv_reg0, slv_reg1) is      -- 寄存器读
  begin
    case slv_reg_read_sel is
      when "10" => slv_ip2bus_data <= slv_reg0;
      when "01" => slv_ip2bus_data <= slv_reg1;
      when others => slv_ip2bus_data <= (others => '0');
    end case;
  end process SLAVE_REG_READ_PROC;

  -- 总线应答信号
  IP2Bus_Data  <= slv_ip2bus_data when slv_read_ack = '1' else
                  (others => '0');
  IP2Bus_WrAck <= slv_write_ack;
  IP2Bus_RdAck <= slv_read_ack;
  IP2Bus_Error <= '0';
end IMP;
```

图 4-9 （续）

尽管使用 IPIF 模块进行设计非常方便，用户在设计 IP 时也可以选择不用 IPIF 模块。对于简单的 OPB 接口的外设，其总线接口代码相当简单，而使用了 IPIF 模块后反而在中断产生、复位逻辑和寄存器读写方面带来了不必要的复杂性。若不使用 IPIF，用户可以修改向导生成的总线接口文件，删除 IPIF 的实例，同时修改任务逻辑模块的端口，以便直接和总线相连，自动生成的寄存器读写电路也需要作相应的修改。IPIF 模块的地址译码功能是必要的，因此在总线接口文件中删除了 IPIF 模块后还需要添加地址译码逻辑，用于生成外设的片选信号。图 4-10 所示为一个不使用 IPIF 模块的总线接口代码片断。

```
entity ip_name is
  port
  (    -- 总线接口
    OPB_Clk          : in  std_logic;
    OPB_Rst          : in  std_logic;
    Sl_DBus          : out std_logic_vector(0 to C_OPB_DWIDTH - 1);
    Sl_errAck        : out std_logic;
    Sl_retry         : out std_logic;
    Sl_toutSup       : out std_logic;
    Sl_xferAck       : out std_logic;
    OPB_ABus         : in  std_logic_vector(0 to C_OPB_AWIDTH - 1);
    OPB_BE           : in  std_logic_vector(0 to C_OPB_DWIDTH/8 - 1);
    OPB_DBus         : in  std_logic_vector(0 to C_OPB_DWIDTH - 1);
    OPB_RNW          : in  std_logic;
    OPB_select       : in  std_logic;
    OPB_seqAddr      : in  std_logic;
    IP2INTC_Irpt     : out std_logic
  );
end entity aes_cipher;

architecture IMP of ip_name is
  -- 计算外设地址范围的高低地址不同部分的起始位置,用于地址译码
  function Addr_Bits (x, y : std_logic_vector(0 to C_OPB_AWIDTH - 1))
    return integer is
    variable addr_nor : std_logic_vector(0 to C_OPB_AWIDTH - 1);
  begin
    addr_nor := x xor y;
    for i in 0 to C_OPB_AWIDTH - 1 loop
      if addr_nor(i) = '1' then return i;
      end if;
    end loop;
    return(C_OPB_AWIDTH);
  end function Addr_Bits;
  constant C_AB : integer := Addr_Bits(C_HIGHADDR, C_BASEADDR);
  -- 片选信号
  signal Bus2IP_CS : std_logic;

begin
  USER_LOGIC_I : entity user_logic    -- 用户逻辑实例
    port map
    (
      Bus2IP_Clk       => OPB_Clk,
      Bus2IP_Reset     => OPB_Rst,
      IP2Bus_IntrEvent  => IP2INTC_Irpt,
      Bus2IP_Data       => OPB_DBus,
      -- 低位地址,用于选择寄存器
      Bus2IP_Addr => OPB_ABus(C_OPB_AWIDTH - 5 to C_OPB_AWIDTH - 3),
      Bus2IP_CS => Bus2IP_CS,
      Bus2IP_RnW       => OPB_RNW,
      IP2Bus_Data      => Sl_DBus,
      IP2Bus_Ack       => Sl_xferAck
```

图 4-10　不使用 IPIF 模块的总线接口代码框架

```
    );

  pselect_I : entity pselect        -- 地址译码
    generic map (
      C_AB   => C_AB,
      C_AW   => C_OPB_AWIDTH,
      C_BAR  => C_BASEADDR)
    port map (
      A       => OPB_ABus,
      AValid  => OPB_select,
      CS      => Bus2IP_CS);

  -- 不必要的总线应答信号
  Sl_errAck     <= '0';
  Sl_retry      <= '0';
  Sl_toutSup    <= '0';

end IMP;
```

图 4-10 （续）

4.4.3 基于 Altera FPGA 的外设接口设计实例

Altera 的 SOPC 设计工具没有提供类似于 IPIF 的模块，因此用户必须自己编写外设的接口电路与功能逻辑模块代码。Avalon 总线的时序比 OPB 更为简单，外设接口电路也比较简单。下面通过两个外设实例介绍外设接口电路的设计。

第一个外设实例是 4 位七段数码显示译码器。图 4-11 所示为其代码片断，重点显示总线接口电路。

```
entity seven_seg is   -- 外设模块
port (
    -- 总线端口
    -- 只写外设,不需要读使能与读数据端口
    clk             :  in  std_logic;
    chipselect      :  in  std_logic;
    reset_n         :  in  std_logic;
    address         :  in  std_logic_vector(1 downto 0);
    write           :  in  std_logic;
    writedata       :  in  std_logic_vector(8 downto 0);
    -- 数码管输出端口
    dig_sel_n       :  out std_logic_vector(3 downto 0);
    seg_out         :  out std_logic_vector(7 downto 0)
);
end seven_seg;

architecture alg of seven_seg is
-- 输入寄存器
signal dig0, dig1, dig2, dig3 : std_logic_vector(8 downto 0);
-- 其他信号定义
```

图 4-11 4 位七段数码显示译码器接口电路代码

```
...

begin
    process(clk, reset_n) -- 总线接口电路,写寄存器
    begin
        if reset_n = '0' then
            ...    -- 寄存器清零
        elsif clk'event and clk = '1' then
            if chipselect = '1' and write = '1' then
                case address is   -- 按照地址写入不同的寄存器
                    when "00" => dig0 <= writedata;
                    when "01" => dig1 <= writedata;
                    when "10" => dig2 <= writedata;
                    when "11" => dig3 <= writedata;
                end case;
            end if;
        end if;
    end process;

-- 任务逻辑电路
...

end;
```

图 4-11　(续)

第二个外设实例是 PS2 键盘接口电路,如图 4-12 所示。键盘每输入一个字节后,电路会产生一个中断。请注意中断生成电路的编写。Nios Ⅱ处理器仅支持电平中断,因此中断信号产生后必须保持到处理器响应了这个中断之后。这里使用了一个技巧:当处理器响应了中断,会运行终端服务程序,而终端服务程序会读取输入的字节,这时片选(chipselect)信号有效,中断生成电路根据这一条件清除中断信号。这一过程在硬件中自动完成,不需要软件处理。

```
entity ps2_keyboard is   -- 外设模块
port(
    -- 总线端口,只读
    clk             : in  std_logic;
    chipselect      : in  std_logic;
    reset_n         : in  std_logic;
    address         : in  std_logic;
    read            : in  std_logic;
    readdata        : out std_logic_vector(7 downto 0);
    irq             : out std_logic;
    -- 键盘数入端口
    ps2_data        : in  std_logic;
    ps2_clk         : in  std_logic
);
end ps2_keyboard;

architecture behavior of ps2_keyboard is
```

图 4-12　PS2 键盘接口电路代码

```
-- 状态定义
type StateType is (...);
signal state, old_state : StateType;

-- 串并转换结果
signal byte_ok : std_logic_vector(7 downto 0);
signal irq_tmp : std_logic;

-- 其他信号定义
...

begin
    -- 任务逻辑，产生 byte_ok
    ...

    interrupt: process(clk, reset_n) -- 中断生成
    begin
        if reset_n = '0' then
            irq_tmp <= '0';
            old_state <= wait_for_start;
        elsif clk'event and clk = '1' then
            if old_state = parity_ok and state = wait_for_start then
                irq_tmp <= '1';
            elsif chipselect = '1' then
                irq_tmp <= '0';
            end if;
            old_state <= state;
        end if;
    end process;

    -- 只有一个寄存器，直接输出即可
    readdata <= byte_ok;
    irq <= irq_tmp;

end behavior;
```

图 4-12 （续）

4.5 基于 Altera FPGA 的嵌入式系统硬件设计

4.5.1 SOPC Builder 简介

SOPC Builder 工具是 Altera 公司提供的一个灵活、方便的系统设计工具，它利用一个组件库搭建基于总线的系统。用户从组件库中挑选所需的组件，配置组件参数，然后 SOPC Builder 自动生成总线互连逻辑，并将参数化后的组件实例连接起来，形成一个完整的可编程片上系统模块。SOPC Builder 可以快速地开发定制的方案，重建已经存在的方案，并为

其添加新的功能，提高系统的性能。通过自动集成系统组件，SOPC Builder 允许用户将工作的重点集中到系统级的设计开发，而不是烦琐的组件装配工作。

SOPC Builder 提供了一个强大的平台，用于组建一个在模块级和组件级定义的系统。SOPC Builder 的组件库包含了微处理器、内存接口、总线桥以及一些常用的外设接口等一系列的组件，用户还可简单地创建定制的 SOPC Builer 组件。

图 4-13 直观地显示了 SOPC Builder 的用户界面。

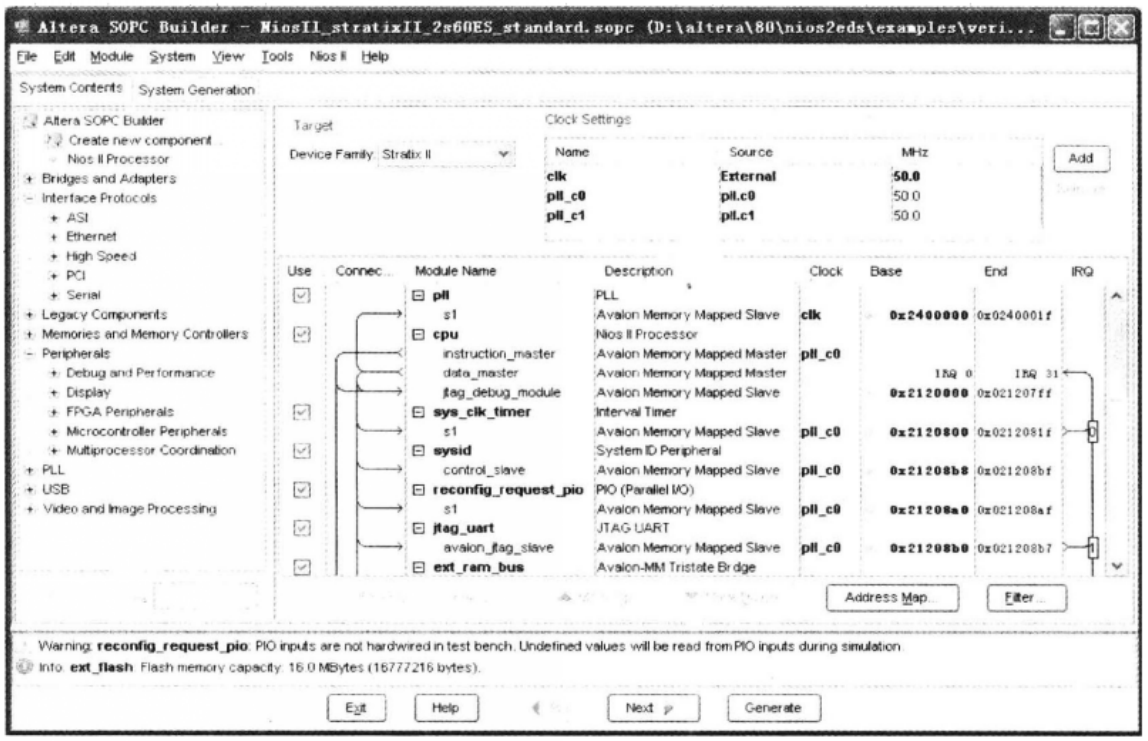


图 4-13　SOPC Builder 用户界面

从用户的角度来看，SOPC Builder 是一个能够生成复杂硬件系统的工具。从内部结构来看，SOPC Builder 包含两个主要部分：一个图形用户界面和一个系统生成程序，如图 4-14 所示。

SOPC Builder 用户图形界面提供管理 IP 模块、配置系统和报告错误等功能。用户通过图形界面设计系统时，所有的设置都保存在一个系统描述文件中。

用户通过 SOPC Builder 用户图形界面完成设计之后，单击图 4-13 中的 Generate 按钮，则启动了系统生成程序。系统生成程序执行了大量的功能，创建了几乎所有的 SOPC Builder 输出文件（HDL 代码、C 程序的头文件、库文件和仿真文件等）。

SOPC Builder 集成在 Quartus Ⅱ 软件中，可以通过 Quartus Ⅱ 的界面启动。SOPC Builder 所生成的系统模块输出给 Quartus Ⅱ，然后用户利用 Quartus Ⅱ 把所创建的 SOPC 系统与其他模块，例如 PLL 等集成，经 Quartus Ⅱ 编译后就产生 FPGA 的配置文件。

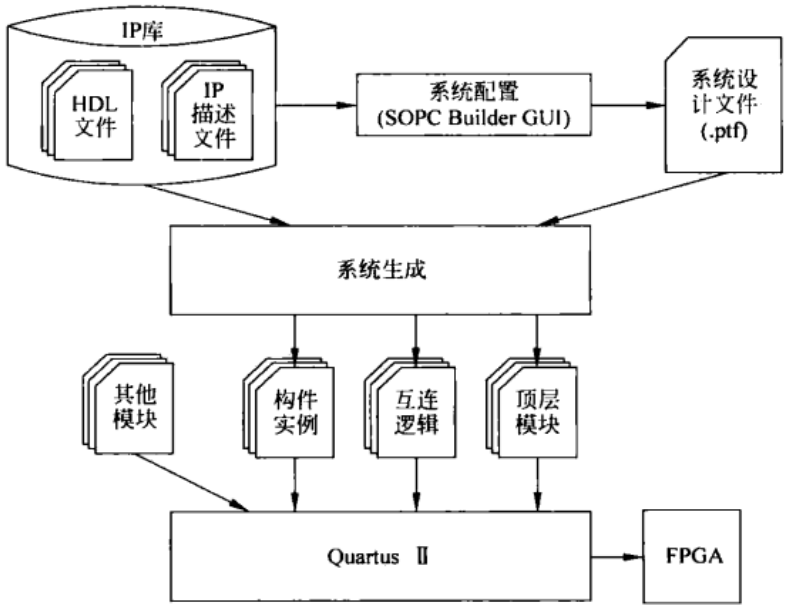


图 4-14 基于 Altera FPGA 的嵌入式系统硬件设计

4.5.2 SOPC Builder 设计流程

SOPC Builder 可看作是以 IP 库为输入，集成后的 SOPC 系统为输出的工具。SOPC Builder 设计过程有三个主要步骤。

1. 组件开发

SOPC Builder 的 IP 模块包含由 IP 开发人员提供的硬件描述，如 RTL 代码、原理图或 EDIF 及其软件，如 C 源代码、头文件等。高级 IP 模块可能还会包含一个相关的用户图形界面，一个生成程序，和其他支持系统参数化生成的程序。在完成 IP 模块的硬件和软件实现后，需要利用 SOPC Builder 中的组件编辑器建立该 IP 模块的描述文件，从而把这个 IP 模块添加到 SOPC Builder 的 IP 模块库中。图 4-15 示出了组件编辑器的界面。

2. 系统集成

用户创建和编辑一个新的系统时，首先要从库中选择一些 IP 模块，并逐个地配置这些 IP 模块，然后设置整个系统的配置，例如指定地址映射和主/从端口连接等，如图 4-16 所示。在这个过程中，用户的设置都会保存在系统描述文件中。

3. 系统生成

当用户在完成了 SOPC Builder 中的设计和配置之后，单击 Generate 按钮，或从命令行执行系统生成程序时，系统生成程序就开始运行。系统生成的结果是一系列设计文件，包括各模块的 HDL 代码实例、总线互连逻辑、系统顶层模块和仿真工程文件等。生成的系统模块可以在 Quartus Ⅱ 中编译并下载到 FPGA 中。

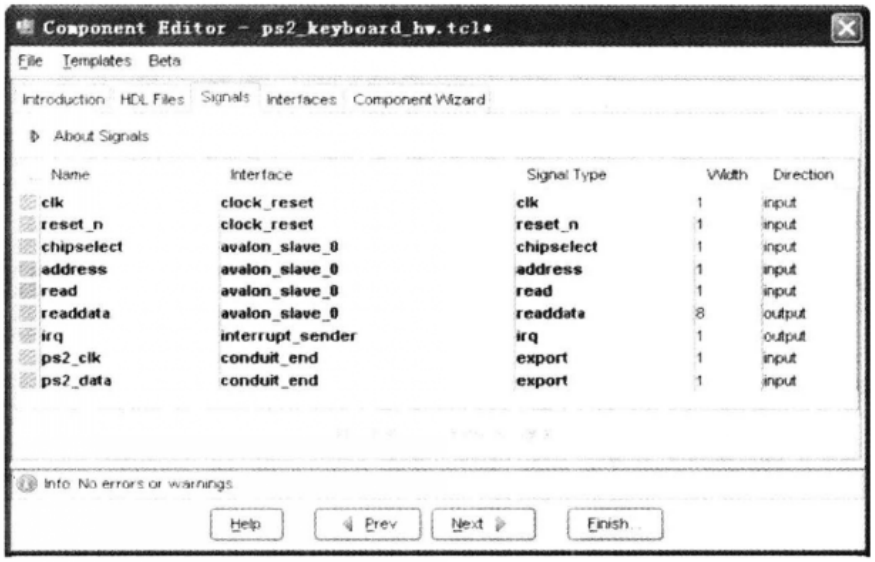


图 4-15　组件编辑器界面

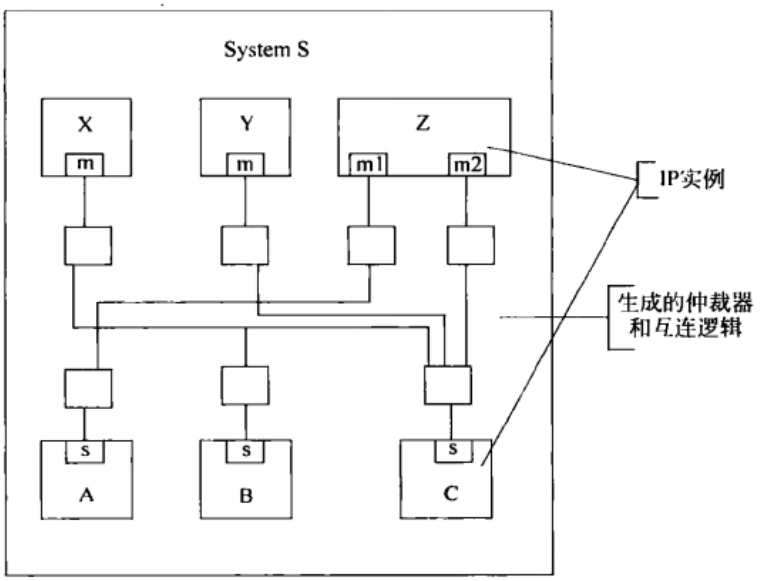


图 4-16　生成后的系统主模块实例[7]

4.6　基于 Xilinx FPGA 的嵌入式系统硬件设计

4.6.1　Platform Studio 简介

Xilinx 公司也提供了一个嵌入式系统设计工具，称为 Platform Studio。与 SOPC Builder 类似，Platform Studio 也包含一个组件库，用户通过图形界面组件库挑选所需的组件，配置组件参数，以及组件间的连接方式。Platform Studio 工具会自动生成互连逻辑，并

将组件实例连接起来，形成一个完整的可编程片上系统。

与 SOPC Builder 工具的不同之处在于 Platform Studio 是一个顶层的开发平台。完整的嵌入式系统设计均可在 Platform Studio 的界面中完成。除了搭建系统外，通过 Platform Studio 的界面还能调用 ISE 工具编译硬件，调用 GNU 工具编译软件，并将硬、软件下载到 FPGA 中运行。

图 4-17 示出了 Platform Studio 的用户界面。

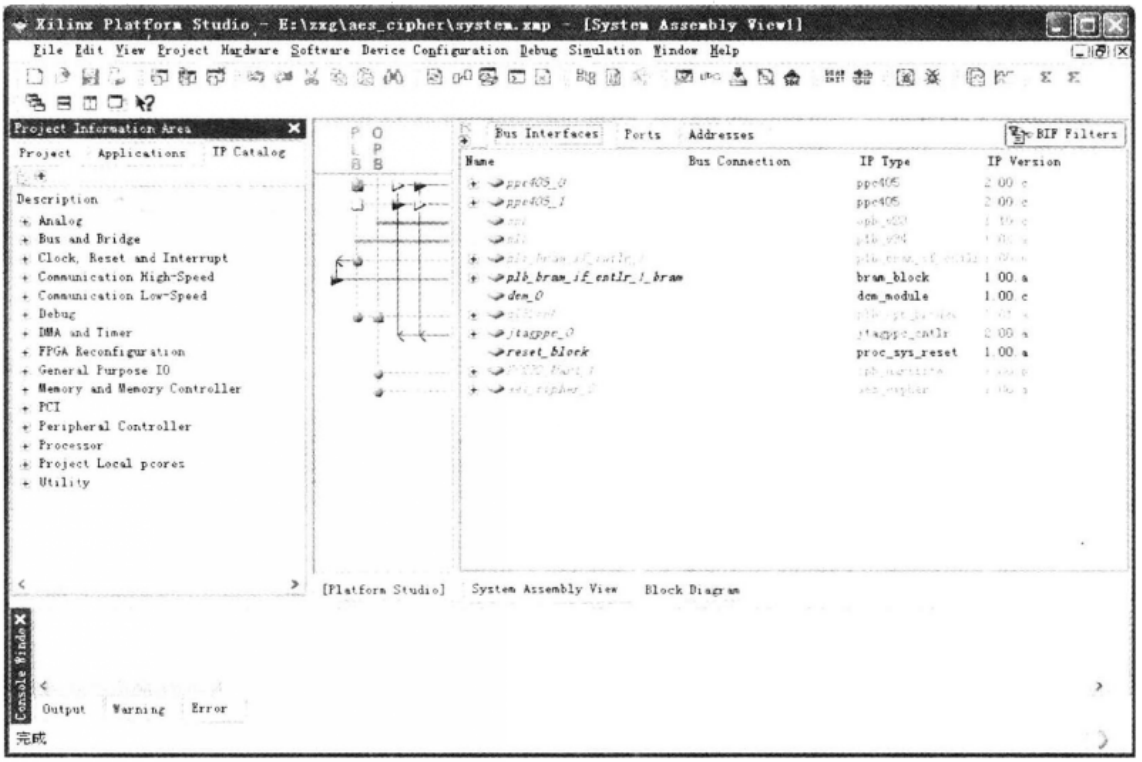


图 4-17　Xilinx Platform Studio 的界面

4.6.2　Platform Studio 设计流程

利用 Platform Studio 工具设计嵌入式系统可分为以下步骤。

1. 创建基本系统

在 Platform Studio 中包含一个基本系统创建向导(base system builder，BSB)，可以帮助用户快速有效地创建一个嵌入式系统的初始设计，用户在此基础上再完成后续的设计。在 BSB 中，用户首先选择一款开发板，然后配置处理器、片外与片内的外设，最后还可以选择一些程序样例。BSB 会自动按照用户的选择搭建起系统。

图 4-18 是基本系统创建向导的界面。

2. 创建自定义外设

如果 Platform Studio 自带的外设库不能满足需要，那么用户还可以创建自定义的外

设，并把它加入到 Platform Studio 的组件目录中。Xilinx 提供了外设创建向导帮助用户完成这一任务。外设创建向导的使用已在 4.4.2 节做了介绍。

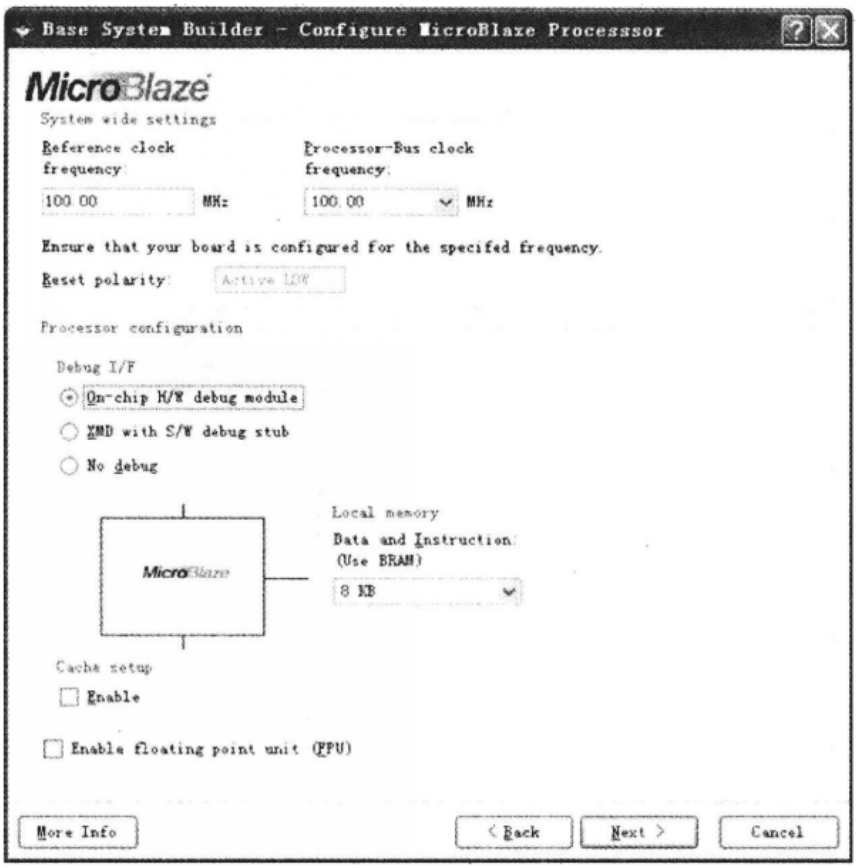


图 4-18　基本系统创建向导

3. 配置系统

在基本系统的基础上，用户还可以在 Platform Studio 的界面中修改系统的配置，包括添加或删除组件、配置组件属性、设置组件间的连接方式、设置软件属性等，如图 4-17 所示。在这个过程中，用户的设置会保存到一组系统描述文件，包括硬件描述文件(MHS)、软件描述文件(MSS)、用户约束文件(UCF)和下载命令文件等。

4. 系统生成

在设计完成后，用户通过 Platform Studio 的界面能够根据系统描述文件生成硬件平台和板级支持包(BSP)。在 Platform Studio 的界面中还能调用 ISE 工具编译硬件，调用 GNU 工具编译软件，并将生成的系统下载到 FPGA 运行，同时还能对系统进行仿真和调试。

图 4-19 显示了完整的 XPS 软硬件开发流程。

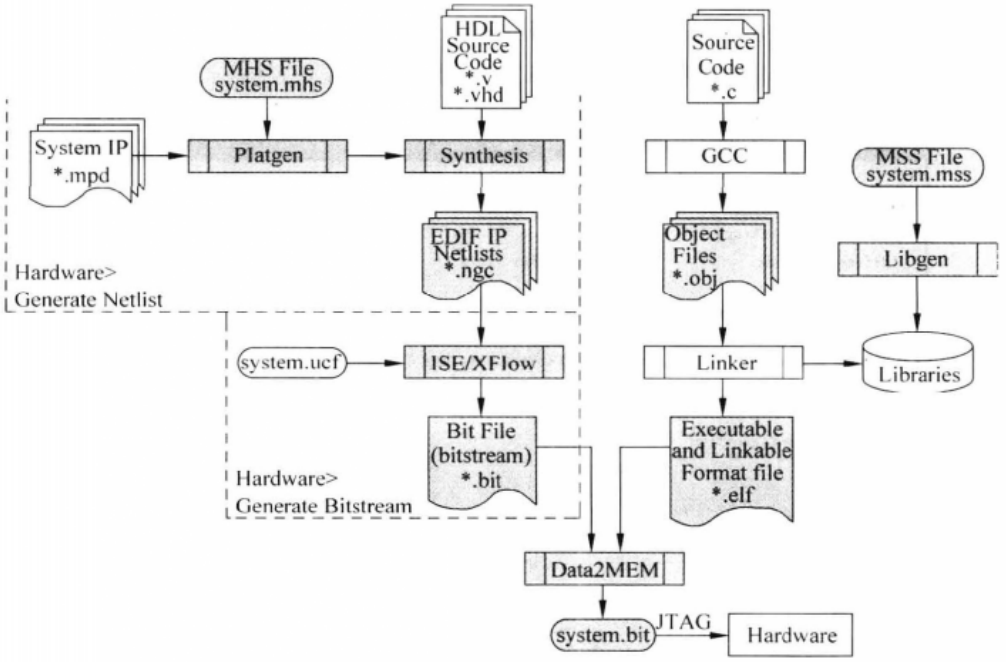


图 4-19 Platform Studio 开发流程

习题

1. 简述 FPGA 在嵌入式系统中的作用，以及采用 FPGA 作为核心器件实现嵌入式系统的优缺点。

2. 比较 Avalon 总线和 CoreConnect 总线的差异以及各自的优缺点。

3. 比较 Altera SOPC Builder 和 Xilinx Platform Studio 的差异以及各自的优缺点。

4. 查阅 OPB 总线相关的技术资料，根据 4.4.2 节由 Xilinx 外设创建向导生成的用户自定义任务逻辑模块代码画出对应的总线读写时序图。

5. 在 SOPC Builder 的组件编辑器中可以为 IP 定义固定的总线读写延时，这样在用户逻辑代码中就不必产生总线应答信号。常见的定义方式是读延时为 1 个时钟周期，而写延时为 0 个周期。根据 Avalon 总线的读写时序图说明，对于用户自定义的同步读写 IP 核，为何 1 个周期的读延时是必需的？

参考文献

[1] 彭澄廉，周博等. 挑战 SoC——基于 Nios 的 SOPC 设计与实践. 北京：清华大学出版社，2004

[2] Navanee Sundaramoorthy. Configurable chips improve FPGA flexibility. http://www.eetasia.com/ARTICLES/2007OCT/PDF/EEOL_2007OCT16_PL_EMS_TA.pdf

[3] Altera Corp. Nios Ⅱ Processor Reference Handbook, May, 2007: 2

[4] Altera Corp. Quartus Ⅱ Version 8.0 Handbook, May.2008, 4: 2-3

[5] Altera Corp. Avalon Interface Specifications, Oct, 2008: 3-10

[6] IBM Corp. CoreConnect Bus Architecture White Paper, 1999

[7] Altera Corp. Sopc Builder PTF File Reference Manual, Version 1.1, Sep. 2002: 33

第5章

基于FPGA的嵌入式系统软件开发

5.1 嵌入式系统软件开发概述

嵌入式系统有着广泛的应用领域，其功能的复杂程度差别也很大。对于简单的嵌入式系统，例如电子门锁和计算器，一般采用简单的4位或8位微控制器(micro controller unit，MCU)作为处理单元，存储容量很小，其软件开发大多使用汇编语言，以达到高速度和低资源占用的要求。另一方面，如今高档智能手机的硬件处理能力和软件功能均已接近微型计算机，其软件开发环境也和通用计算机的开发相似，可选择各种高级语言，例如C/C++、Java等，并且有丰富的函数库可以使用，在运行时由操作系统负责资源管理。对于中低复杂程度的嵌入式系统，其软件一般采用C语言来开发。软件程序直接控制硬件操作，并不需要操作系统的管理。

为了方便软件的开发，通常将计算机软件分为多个层次，如图5-1所示。

(1) 硬件抽象层隐藏了各种硬件设备的差异，为软件提供统一的接口。

(2) 操作系统对硬件资源提供管理，使得应用程序不必关心硬件资源的操作细节。

(3) 函数库为应用程序的开发提供了各种可复用的基础功能，包括文件系统、图形系统、网络通信等。操作系统提供的服务也是通过库的包装提供给编程语言使用的。

(4) 最上层是应用程序，提供用户要求的各种功能或者用户自己编写代码实现所需要的应用功能。

在基于FPGA的嵌入式系统中，系统功能可以通过FPGA硬件编程实现，也可通过软件编程实现。硬件实现性能较高，适合于计算密集型的任务；而软件实现则比较方便灵活，适合于控制类的任务。在系统设计时，需要统一考虑系统的软硬件功能实现，以满足系统的成本、性能、体积、功耗等各项指标。这种设计方法称为软硬件协同设计，其过程如图5-2所示。首先根据一定的准则对系统的功能描述进行软硬件划分，决定哪些功能由软件实现，哪些功能由硬件实现；接下来分别进行软件设计和硬件设计，通过软硬件协同仿真验证其功能和性能等指标；若不满足设计要求，可对软硬件划分进行调整；最后，硬件综合后得到FPGA位流配置文件，软件编译后得到可执行程序，二者集成起来形成完整的系统。

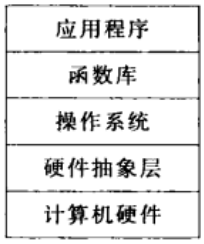


图 5-1　计算机软件层次

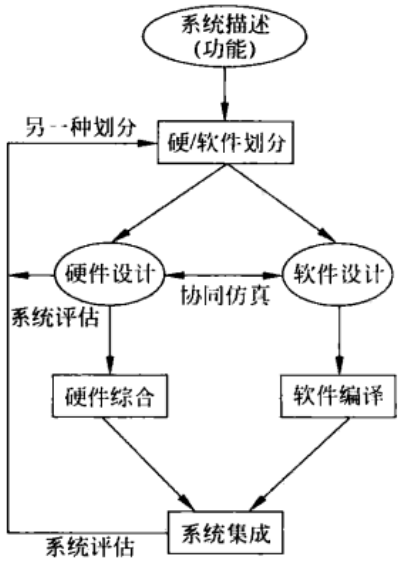


图 5-2　硬软件协同设计流程

5.2　嵌入式系统软件结构

嵌入式系统通常重复地执行特定的功能，最简单的软件结构是轮转结构，用一个主循环轮流检测系统的各个外围设备，进行数据的采集、处理和输出工作，如图 5-3 中的代码所示。

```
int main()
{
    while(1) {
        if (外设 1 需要服务) {
            处理外设 1 的数据
        }
        if (外设 2 需要服务) {
            处理外设 2 的数据
        }
        ...
        if (外设 n 需要服务) {
            处理外设 n 的数据
        }
    }
}
```

图 5-3　轮转结构

这种结构只适合于较简单的系统，完成一次循环的时间不宜过长。例如当系统处理音频数据时，主循环运行周期必须小于音频采样周期(通常只有几十微秒)，否则就会严重影响声音质量。对于比较复杂的应用，轮转结构难以满足系统的性能要求。

采用中断服务可以有效地解决轮转结构的问题，这就是图 5-4 所示的前后台结构。中断服务程序作为系统的前台任务，能够及时响应设备的请求。但一次中断服务的运行时间不能过长，因此较复杂的数据处理任务放在后台主循环中执行，而中断服务程序只处理数据的输入与输出。

对于更为复杂的系统，可以将后台任务分解成多个任务，采用实时操作系统进行调度，如图 5-5 所示。操作系统能够按照优先级来调度任务，更好地保证系统的实时性。

上述三种软件结构都有各自的适用范围。在进行嵌入式系统设计时，在能够满足系统设计要求的情况下，应选择较简单的程序结构。表 5-1 对三种软件结构的特点做了总结比较。

```
void isr1()     // isr: interrupt service routine
{
    外设 1 的中断服务
}
void isr2()
{
    外设 2 的中断服务
}
...
void isrn()
{
    外设 n 的中断服务
}
int main()
{
    while(1) {
        后台数据处理
    }
}
```

图 5-4　中断服务采用的前后台结构

```
void isr1()
{
    外设 1 的中断服务
}
void isr2()
{
    外设 2 的中断服务
}
...
void isrn()
{
    外设 n 的中断服务
}
void task1()
{
    任务 #1
}
void task2()
{
    任务 #2
}
...
void taskm()
{
    任务 #m
}
```

图 5-5　实时操作系统结构

表 5-1　三种软件结构的比较

特征	轮转结构	前后台结构	实时操作系统结构
优先级	无	中断服务有优先级，后台任务无优先级	中断服务有优先级，操作系统任务有优先级
实时性	差	较好	好
复杂性	很简单	较复杂，需考虑前后台代码对共享数据的并发访问	复杂

5.3　嵌入式系统软件开发工具

在通用计算机上，软件开发和软件运行的计算机环境通常是一样的。而嵌入式系统往往不具备足够的软硬件资源用来进行软件开发，因此需要使用一台通用计算机作为宿主机(host)进行软件开发，编写的程序经过编译、连接后，生成目标计算机(target)上的可执行程序，加载到嵌入式系统上运行。

GNU(GNU is not unix)工具链是一组由 GNU 项目产生的开源编程工具，包括汇编器、编译器和连接器等。这些工具形成了一条工具链，即串行使用的一组工具，能够将高级语言编写的代码转换成二进制可执行程序。由于其开源的特性，GNU 工具链能够方便地移植到各种不同的处理器，支持宿主机和目标机分离的交叉编译方式。它是各种嵌入式系

统的开发中使用最广泛的交叉编译工具链。

交叉编译工具链的编译过程如图 5-6 所示。用高级语言编写的源程序首先经过交叉编译器生成目标文件，连接器将多个目标文件和库文件连接起来，生成可执行程序。例如一个简单的 C 语言源程序 hello_world. c 经过 Nios Ⅱ的交叉编译器 nios2-elf-gcc 编译后得到目标文件 hello_world. o。图 5-7 是目标文件反汇编后的结果，包含有源程序的对照。

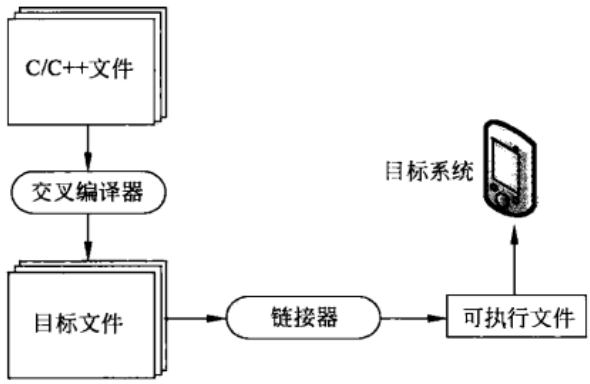


图 5-6 交叉编译工具链

注意目标文件中代码地址是从 0 开始定义的，程序中调用的 printf 等外部函数的地址尚未确定。

连接程序将目标文件同 C 语言标准库连接起来，得到可执行程序 hello_world. elf。图 5-8 是可执行程序反汇编的片段，main 函数和 printf 的地址已经确定。

```
00000000 <main>:
int main()
{
   0:    defffd04    addi    sp, sp,
-12
   4:    dfc00215    stw ra,8(sp)
   8:    df000115    stw fp,4(sp)
   c:    df000104    addi    fp,sp,4
  printf("Hello from Nios Ⅱ!\n");
  10:    01000034    movhi    r4,0
  14:    21000004    addi    r4,r4,0
  18:    00000000    call    0 <main>
  return 0;
  1c:    0005883a    mov r2,zero
}
  20:    dfc00217    ldw ra,8(sp)
  24:    df000117    ldw fp,4(sp)
  28:    dec00304    addi    sp,sp,12
  2c:    f800283a    ret
```

图 5-7 hello_world. o 的反汇编结果

```
00100038 <main>:
int main()
{
  100038:    defffd04    addi    sp,sp, -12
  10003c:    dfc00215    stw ra,8(sp)
  100040:    df000115    stw fp,4(sp)
  100044:    df000104    addi    fp,sp,4
  printf("Hello from Nios Ⅱ!\n");
  100048:    01000474    movhi    r4,17
  10004c:    212ebf04    addi    r4,r4, -17668
  100050:    01000ec0    call    1000ec <printf>
  return 0;
  100054:    0005883a    mov    r2,zero
}
  100058:    dfc00217    ldw ra,8(sp)
  10005c:    df000117    ldw fp,4(sp)
  100060:    dec00304    addi    sp,sp,12
  100064:    f800283a    ret
```

图 5-8 hello_world. elf 的反汇编结果

编译器将目标文件中的代码和数据分为多个段(section)，通常包括 text(代码段)、rodata(只读数据段)、rwdata(可读写数据段)、bss(未初始化数据段)等。连接器的主要功

能是将各个目标文件中相同段合并起来，并为其分配地址，同时填充目标文件中尚未确定的外部函数或变量的地址。嵌入式系统往往包含多个存储器设备，各自具有不同的地址范围。连接器需要知道系统的存储器配置信息，如此才能为代码和数据分配地址空间，这些信息是由连接脚本(linker script)描述的。图 5-9 所示的连接脚本描述了一个 SOPC 系统的内存地址范围，以及各个段的地址分配方式。

```
MEMORY                  /* 内存地址范围 */
{
    reset : ORIGIN = 0x00000000, LENGTH = 32
    onchip_mem : ORIGIN = 0x00000020, LENGTH = 20448
    sram : ORIGIN = 0x00100000, LENGTH = 524288
}
ENTRY( _start )         /* 程序入口 */
SECTIONS
{
    .entry :            /* 入口地址分配到 reset */
    {
        KEEP ( *(.entry))
    } > reset

    .exceptions :       /* 中断向量分配到 onchip_mem */
    {
    ...
    } > onchip_mem

    .text :             /* 代码分配到 sram */
    {
    ...
    } > sram

    .rodata :           /* 只读数据分配到 sram */
    {
       ...
    } > sram
    ...
}
```

图 5-9　连接脚本片段

连接脚本的内容比较复杂，但软件设计工具能够根据系统的硬件配置自动生成连接脚本，不必由设计人员手工编写。

5.4　自定义外设驱动设计

在通用计算机上的软件开发中，普通程序员很少会接触到驱动程序的设计。而在基于 FPGA 的嵌入式系统中，情况则有所不同。FPGA 中用户自定义 IP 模块通常作为外设与 CPU 相连，实现输入输出和数据处理等功能。设备驱动程序是访问和控制设备的基础软件，用户自定义的外设需要配有驱动程序，才能在软件开发中使用。因此在基于 FPGA 的嵌入式系统的软件开发中，驱动程序的设计有着重要的意义。

5.4.1 设备驱动程序的层次结构

每个设备都定义了一组寄存器，作为与软件的数据通信与控制接口。它一般包括数据寄存器、状态寄存器和控制寄存器等。软件程序通过读写不同的寄存器，实现对外设的数据通信与控制，如图 5-10 所示。

驱动程序的功能可按照三个层次来定义，如图 5-11 所示。最底层是寄存器层。读写设备的寄存器是驱动程序必须实现的基础功能，为此驱动程序应提供一组宏或函数，它们定义了设备寄存器的名称以及读写各个寄存器的基本操作。第二层是功能层，在第一层的基础上进一步实现设备的操作功能。操作系统等上层软件为设备驱动定义了标准的访问接口，为此驱动程序还应实现一个标准接口层，以提供上层软件要求的接口。下面通过两个具体的设计实例来展示驱动程序的设计。

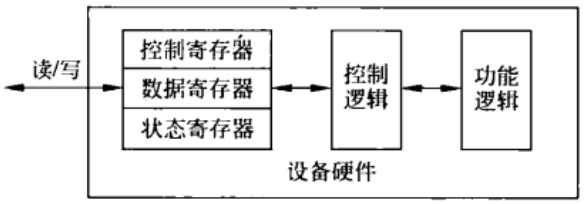


图 5-10 设备寄存器接口

图 5-11 设备驱动程序的层次结构

5.4.2 基于 Altera FPGA 的外设驱动设计实例

Altera 提供了一个 HAL(hardware abstraction layer，硬件抽象层)系统库，它为应用程序和底层硬件的通信提供了一个简单的设备驱动接口。用户设计的设备驱动程序需要符合 HAL API(application program interface)的接口要求。HAL 系统库将应用程序和设备驱动程序清晰地区分开来，如图 5-12 所示。应用程序通过 C 标准库和 HAL API 访问硬件资源；设备驱动程序直接与硬件通信，并通过 HAL API 向应用程序提供访问硬件资源的接口。HAL 将嵌入式系统中的外设分成了几个大类，例如字符模式设备、文件系统、定时器设备和以太网设备等。每一类外设都具有统一的 API 接口，这既为应用程序提供了硬件设备的高层次的抽象，也为设备驱动程序的开发提供了一个规范。

第一个实例是第 4 章 4.4.3 节中 PS2 键盘接口电路的驱动程序。这是一个简单的设备，只提供了一个只读的寄存器，每当键盘输入一个字节后，接口电路会产生一个中断信号。

按照 HAL 规范的要求，设备驱动程序相关的文件放置在几个子目录中，如图 5-13 所示。HAL 目录包含两个子目录，分别存放设备驱动程序的头文件和源代码文件。与 HAL 目录并列的“inc”目录存放定义外设寄存器接口的头文件。

在利用 SOPC Builder 工具搭建系统时，它为每个系统模块都分配了一段地址范围，其起始地址称为基地址。对于外设来说，这段地址就是设备寄存器占用的地址。不同的寄存器相对于外设的基地址具有不同的偏移量。

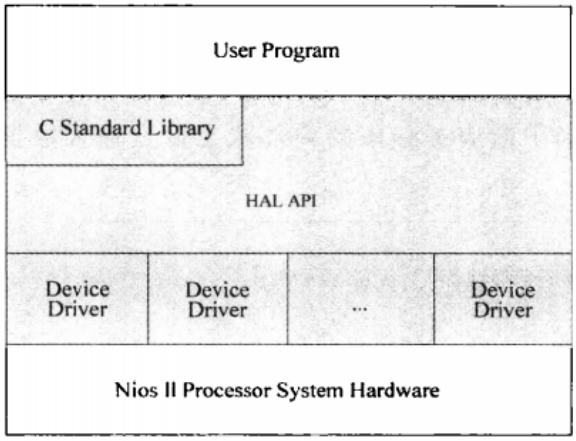


图 5-12　基于 HAL 的软件层次结构

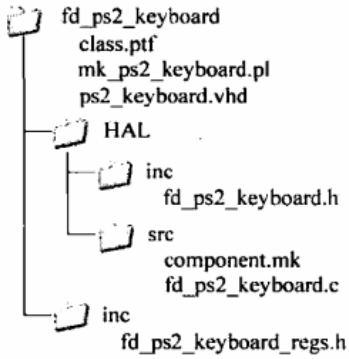


图 5-13　外设驱动程序的目录结构

驱动程序的寄存器层由头文件 fd_ps2_keyboard_regs.h 实现。它定义了一个宏，用来访问设备的数据寄存器。宏的参数是外设的基地址，如图 5-14 所示。

```
#include <io.h>
#define IORD_FD_PS2_KEYBOARD_DATA(base)     IORD(base, 0)
```

图 5-14　键盘接口驱动的寄存器层

功能层实现了驱动程序的主要功能，包括键盘扫描码到 ASCII 码的转换，以及输入数据的缓冲。如图 5-15 所示，功能层的核心是中断服务程序 fd_ps2_keyboard_irq。每次中断到来时，中断服务程序从数据寄存器读入一个字节的扫描码。当一个按键的扫描码全部读入后就可查表得到对应的 ASCII 码，把它存放到输入缓冲区。此外功能层还包括另外两个函数：完成中断服务注册和 HAL 设备注册的初始化函数 fd_ps2_keyboard_init，从输入缓冲区读取 ASCII 码的读字符函数 fd_ps2_keyboard_read。这些函数由 fd_ps2_keyboard.c 源文件实现。

```
/* 中断服务程序 */
static void fd_ps2_keyboard_irq(void* context, alt_u32 id) {
    /* 代码转换,填充数据缓冲区 */
    ...
}
void fd_ps2_keyboard_init(fd_ps2_keyboard_dev* dev, int irq) {
    /* 注册中断服务 */
    if (alt_irq_register(irq, dev, fd_ps2_keyboard_irq) >= 0)
        alt_dev_reg(&dev->dev);     /* 注册 HAL 设备 */
}
int fd_ps2_keyboard_read(alt_fd* fd, char * buffer, int space) {
    /* 得到键盘设备数据指针 */
    fd_ps2_keyboard_dev* dev = (fd_ps2_keyboard_dev*) fd->dev;
    /* 从数据缓冲区中取出字符,并调整缓冲区指针 */
    ...
}
```

图 5-15　键盘接口驱动的功能层

在驱动程序寄存器层定义的函数或宏通过基地址来访问特定的外设。而在功能层，驱动程序往往需要为外设定义一个复杂的数据结构，以实现设备管理和状态存储。功能层的函数通过指向设备数据的指针来访问特定的设备。在本实例中，键盘接口设备的数据结构定义如图 5-16 所示，主要内容包括中断服务程序的内部状态以及输入数据缓冲区。

```
typedef struct {
alt_dev dev;                    // 扩展 HAL 的字符设备
unsigned int base;              // 基地址
/* 中断服务程序状态 */
int scan_len, scan_status;
int mod_status;
/* 输入缓冲区及缓冲区指针 */
char rx_buf[FD_PS2_KEYBOARD_BUF_LEN];
volatile int rx_in;
int rx_out;
} fd_ps2_keyboard_dev;
```

图 5-16 键盘接口驱动的数据结构

在标准接口层，键盘接口驱动程序实现了 HAL 的字符设备接口。字符设备是能够收发字符流的外设。应用程序通过 C 标准库的文件操作函数访问字符设备，也可以将字符型设备指定到标准输入输出，便于 scanf 和 printf 等函数访问。

字符设备由一个 alt_dev 结构来定义，其主要内容是一组函数指针。这些函数实现是字符型设备驱动程序的核心，实现了文件系统操作的各个基本功能。图 5-17 中的代码定义了 alt_dev 数据结构。

```
typedef struct {
alt_llist llist; /* for internal use */
const char* name;
int (*open) (alt_fd* fd, const char* name, int flags, int mode);
int (*close) (alt_fd* fd);
int (*read) (alt_fd* fd, char* ptr, int len);
int (*write) (alt_fd* fd, const char* ptr, int len);
int (*lseek) (alt_fd* fd, int ptr, int dir);
int (*fstat) (alt_fd* fd, struct stat* buf);
int (*ioctl) (alt_fd* fd, int req, void* arg);
} alt_dev;
```

图 5-17 alt_dev 数据结构

一个特定的设备驱动程序不必实现全部的函数，因此 HAL 为每个函数都提供了缺省操作。本实例只提供一个 read 函数，即功能层的读字符函数。驱动程序通过定义一个实例化宏来实现标准接口函数和驱动内部函数间的关联，代码如图 5-18 所示。

5.4.3 基于 Xilinx FPGA 的外设驱动设计实例

本实例为一个数据加密模块设计设备驱动程序。如图 5-19 所示，在硬件模块中包含两个 FIFO 实现数据缓冲，以屏蔽软硬件速度的差异。模块定义了一组寄存器，用来读写

FIFO 和查询 FIFO 状态。

```
#define FD_PS2_KEYBOARD_INSTANCE(name, device)     \
  static fd_ps2_keyboard_dev device = {            \
    {                                              \
      ALT_LLIST_ENTRY,     name## _NAME,           \
      NULL, /* open */                             \
      NULL, /* close */                            \
      fd_ps2_keyboard_read,                        \
      NULL, /* write */                            \
      NULL, /* lseek */                            \
      NULL, /* fstat */                            \
      NULL, /* ioctl */                            \
    },                                             \
    name## _BASE, /* base */                       \
  }
```

图 5-18　键盘接口设备实例化宏

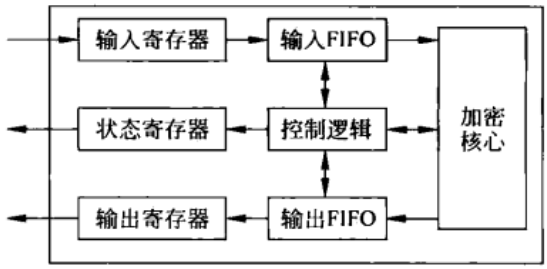


图 5-19　数据加密模块结构框图

驱动程序的寄存器层定义了设备寄存器的基本操作，代码如图 5-20 所示。

```
/* 寄存器地址偏移量 */
#define CIPHER_RX_FIFO_OFFSET              0
#define CIPHER_TX_FIFO_OFFSET              4
#define CIPHER_STATUS_REG_OFFSET           8
/* 状态寄存器位定义 */
#define CIPHER_SR_TX_FIFO_FULL      0x20000000    /* 发送 FIFO 满 */
#define CIPHER_SR_TX_FIFO_EMPTY     0x10000000    /* 发送 FIFO 空 */
#define CIPHER_SR_TX_FIFO_FREE_CNT  0x0FFF0000    /* 发送 FIFO 空闲数 */
#define CIPHER_SR_RX_FIFO_OVERFLOW  0x00008000    /* 接收 FIFO 溢出 */
#define CIPHER_SR_RX_FIFO_FULL      0x00002000    /* 接收 FIFO 满 */
#define CIPHER_SR_RX_FIFO_EMPTY     0x00001000    /* 接收 FIFO 空 */
#define CIPHER_SR_RX_FIFO_DATA_CNT  0x00000FFF    /* 接收 FIFO 占用数 */
/* 寄存器基本操作 */
#define Cipher_mGetStatusReg(BaseAddress) \
    XIo_In32((BaseAddress) + CIPHER_STATUS_REG_OFFSET)
#define Cipher_mSendWord(BaseAddress, Data) \
    XIo_Out32((BaseAddress) + CIPHER_TX_FIFO_OFFSET, (Data))
#define Cipher_mRecvWord(BaseAddress) \
    XIo_In32((BaseAddress) + CIPHER_RX_FIFO_OFFSET)
```

图 5-20　加密模块驱动程序寄存器层

与上一节的键盘接口不同，本实例在硬件中实现数据缓冲，驱动程序就不必再开设数据缓冲区了。由于硬件实现的加密算法速度很快，几乎能够立刻处理完软件发送给它的数据，驱动程序只需用轮询的方式检查设备的状态。此时若采用中断方式反而会加重软件的开销。在驱动程序的功能层实现了数据块的输入输出。以输出函数为例，驱动程序反复查询设备能够接收的数据量，发送数据，直到发送完成或设备不能接收为止，如图 5-21 所示。

```
unsigned int Cipher_Send(Htask * InstancePtr,
              Xuint32 * DataBufferPtr, unsigned int NumWords) {
    unsigned int SentCount = 0;
    unsigned int FreeCount;
    int i;
    while (SentCount < NumWords) {
      FreeCount = Cipher_mTransmitFreeCount(
                  InstancePtr->RegBaseAddress);
        if (FreeCount == 0) break;
        for ( i = 0; i < FreeCount && SentCount < NumWords; i++ )
            Htask_mSendWord(InstancePtr->RegBaseAddress,
                    DataBufferPtr[SentCount++]);
    }
    return SentCount;
}
```

图 5-21 数据块输出函数

Xilinx 没有定义一个类似于 Altera 公司 HAL API 的驱动程序接口规范。驱动程序的设计相对比较自由，若不使用操作系统，就不必设计标准接口层了。

5.5 Altera 与 Xilinx 的软件设计工具

5.5.1 Altera Nios Ⅱ IDE

Altera 公司提供了基于 Eclipse 平台的集成开发环境 Nios Ⅱ IDE，用于 Nios Ⅱ 软核处理器的软件开发。Nios Ⅱ IDE 包含项目管理、源文件编辑、编译和调试等功能。

使用 Nios Ⅱ IDE 进行软件开发的流程如下。

1. 创建应用软件项目

创建项目时需指定对应的硬件系统，如图 5-22 所示。

2. 设置系统库项目

Nios Ⅱ IDE 的每一个应用软件项目都关联着一个系统库项目。在创建应用软件项目时可以同时新建一个系统库项目，也可以重用已存在的系统库项目。系统库项目负责生成 HAL 系统库，包括 C 标准库、硬件系统中各个设备的驱动程序、HAL API、系统和设备的初始化函数等。

在系统库的设置页中提供了丰富的设置选项，主要包括是否使用实时操作系统(RTOS)、标准输入输出设备、系统时钟定时器设备、程序各段放置的物理存储器位置等，如图 5-23 所示。

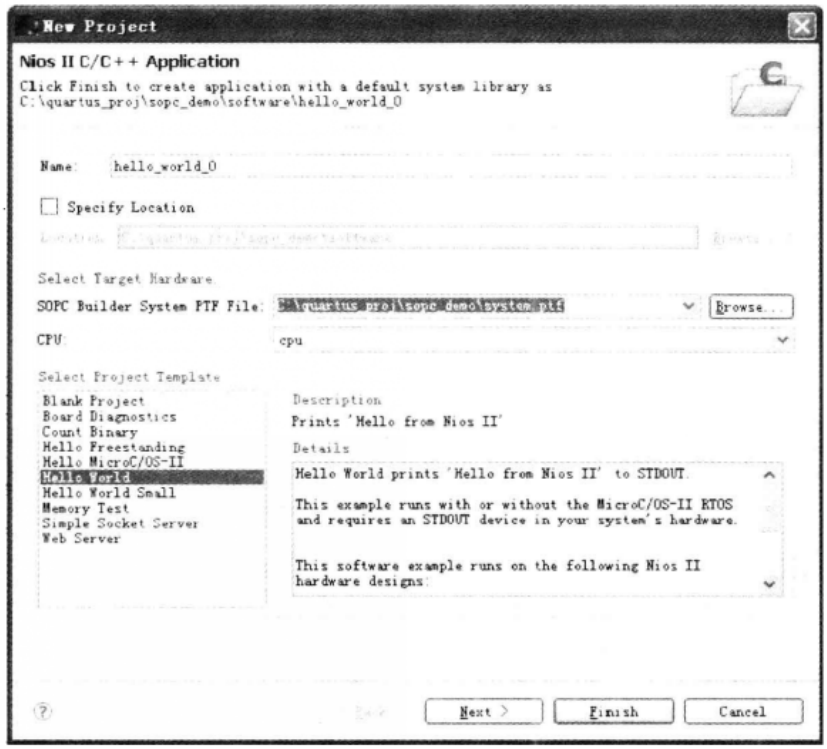


图 5-22　创建 Nios Ⅱ应用软件项目

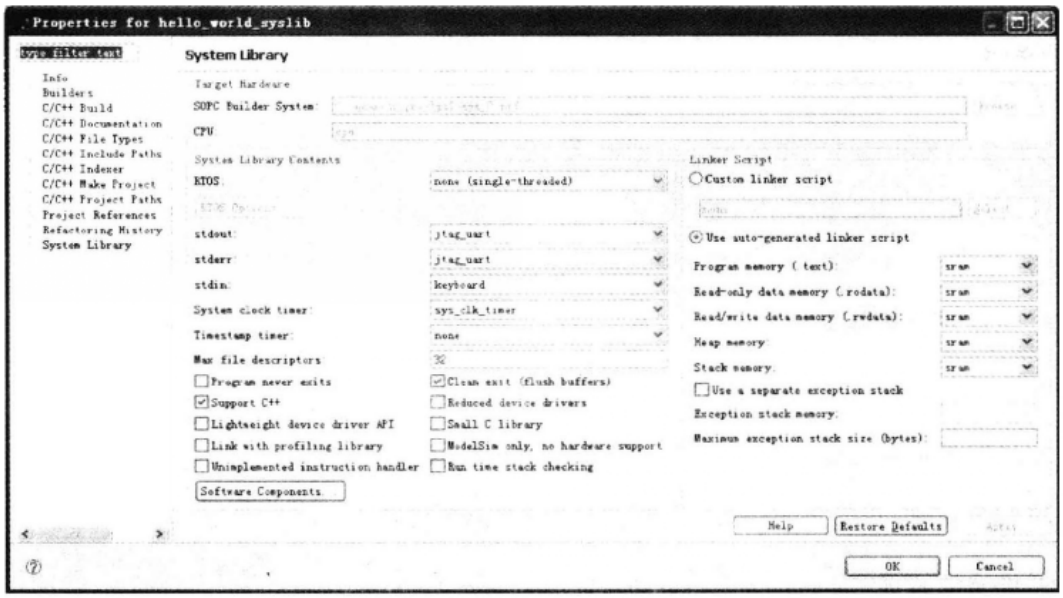


图 5-23　系统库项目设置

3. 编辑和编译源程序

在 Nios Ⅱ IDE 中能方便地创建、导入和编辑 C/C++语言源程序，程序编辑界面如图 5-24 所示。设计完成后可以编译生成在硬件系统上运行的可执行程序。

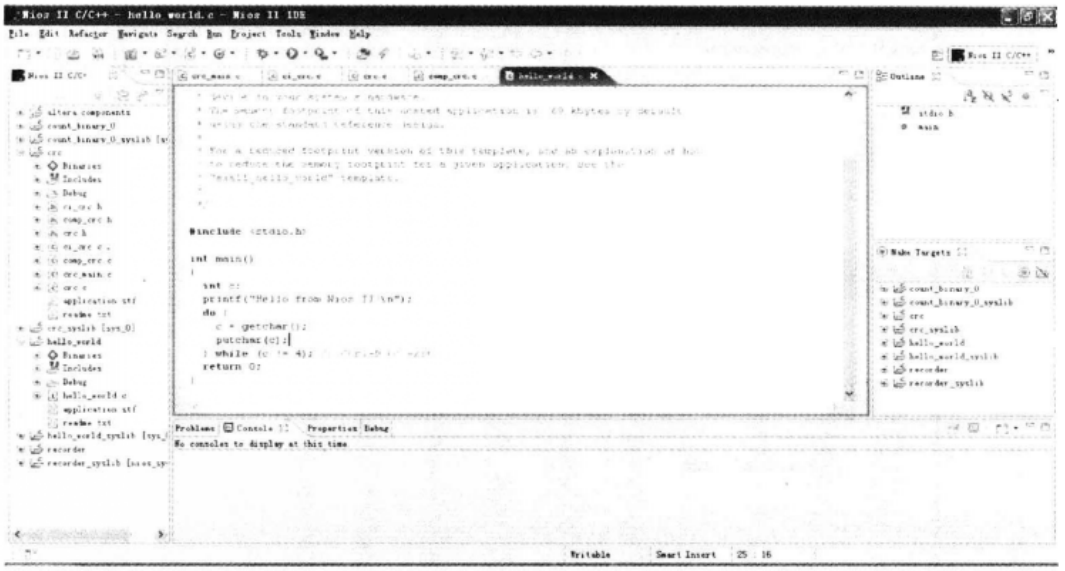


图 5-24 Nios Ⅱ IDE 程序编辑界面

4. 调试/运行程序

应用程序编译完成后，可将其下载到 FPGA 芯片中运行和调试。调试界面如图 5-25 所示。在调试或者运行程序之前，需确保程序对应的硬件系统已经下载到了 FPGA 中。在没有 FPGA 开发版的情况下，应用程序也可以在 Nios Ⅱ 指令集仿真器(instruction set simulator，ISS)中运行。ISS 提供了一个功能有限的硬件仿真系统，包含存储器、UART 和定时器等一些常用的外设。

5.5.2 Xilinx Platform Studio 和 SDK

Xilinx Platform Studio(XPS)同时支持硬件开发和软件开发，如第 4 章的图 4-19 所示。与 Nios Ⅱ IDE 的开发步骤类似，XPS 的软件开发流程主要包括以下步骤。

1. 创建应用软件项目

软件项目与 XPS 中正在设计的硬件系统关联。

2. 设置软件平台选项

相当于 Nios Ⅱ IDE 中的系统库设置，如图 5-26 所示。

3. 编辑源程序

XPS 提供了源程序的编辑和管理功能，如图 5-27 所示。

4. 生成系统库

根据硬件平台和软件平台的设置，生成相应的系统库。其内容包括 C 标准库、硬件系统中各个设备的驱动程序、系统初始化等。

图 5-25　Nios Ⅱ IDE 程序调试界面

图 5-26　软件平台设置

5. 生成应用程序

在生成了系统库之后就可以编译应用程序，从而生成可执行文件。

图 5-27　XPS 软件开发界面

6. 下载运行程序

将可执行文件下载到 FPGA 芯片中运行。同样需要保证 FPGA 的硬件系统已经下载准备完毕。

XPS 不具备软件调试功能，为此 Xilinx 提供了一个 XPS SDK，如图 5-28 所示。它和 Nios Ⅱ IDE 一样是基于 Eclipse 平台，功能也很相似，在此就不重复介绍了。

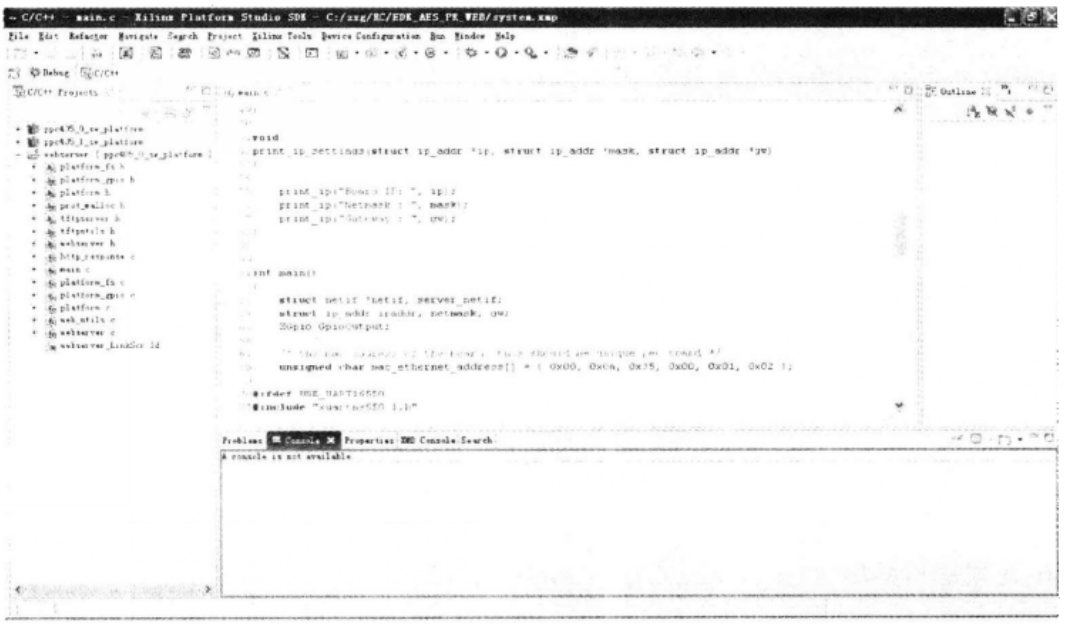


图 5-28　XPS SDK 软件开发界面

习题

1. 简述基于 FPGA 的嵌入式系统软件开发与通用计算机的软件开发的异同。

2. 说明计算机软件层次结构及其相互依赖关系。

3. 比较不同嵌入式系统软件结构的优缺点。在 DE2 板上设计以下应用系统时采用何种结构比较合适？

（1）录音机　　（2）数码摄像机　　（3）Web 服务器

参考文献

[1] David E Simon 著. 陈向群等译. 嵌入式系统软件教程. 北京：机械工业出版社，2005

[2] Cau A，Hale R，Dimitrov J，et al. A Compositional Framework for Hardware/Software Co-design. Design Automation for Embedded Systems，2002：367-399

[3] Altera Corp. Embedded Design Handbook. June，2008

[4] Altera Corp. Nios Ⅱ Software Developer's Handbook. May，2008

[5] Xilinx Inc. Embedded System Tools Reference Manual v9.2i. September，2007

[6] Xilinx Inc. Device Driver Programmer Guide. June，2007

第6章

基于FPGA的可重构系统

本章将主要介绍基于FPGA的可重构系统及其设计方法。首先概述可重构计算，接着对可重构系统及其结构进行分类讨论，然后以部分可重构系统为研究对象，介绍模块化的可重构系统设计方法及其设计流程，最后根据该模块化设计方法给出一个可重构音频信号处理系统设计实例。

6.1 可重构计算概述

在传统的计算系统中，通常采用两种不同的方法来实现算法[1]：一是采用专用集成电路(ASIC)实现方式。这种方法具有很高的执行速度和运算精度。但ASIC一经设计，其功能便不能发生变化，要想实现不同的算法必须重新设计电路，开发周期及成本较高。而且ASIC只有在大批量生产时才能显出其成本的优越性。二是使用通用微处理器实现方式。该方法灵活性高，一旦需求发生变化，可通过软件指令改变系统的功能。但指令的串行执行、内存访问带宽瓶颈等因素使得通用微处理器的性能往往难以满足实际需求，并且基于单核或多核处理器的软件实现对于具备潜在并行特性的复杂算法不能提供很成熟的技术支持。

近年来随着以FPGA为代表的可重构逻辑器件的发展，可重构计算(reconfigurable computing)已成为系统结构领域的研究热点之一。作为一种全新的计算方式，可重构计算结合了通用化和专用化的优点，它利用FPGA可多次重配置逻辑单元功能和互连的特性，因此可以根据应用需求动态配置电路的实现形式，从而兼具硬件实现的高性能和软件实现的灵活性[2]。

可重构计算的概念早在20世纪60年代就已经被提及。美国加利福尼亚大学的Geraid Estrin首先提出可重构的概念，并研制了原型系统。该系统由非柔性但可编程的处理器和柔性的由程序控制重构的数字逻辑部件两部分组成[3]。尽管系统软件和硬件的抽象层次不高，但两者均可编程重构。由于当时实现技术尚不完善，Estrin研制的系统只是设计理念上的粗略近似，但这种结构奠定了以后可重构系统的核心基础。

1985年Xilinx公司开发出世界上第一片FPGA芯片，并在实践中获得了很好的应用效果。到20世纪90年代初，出现了一些以FPGA为核心开发的面向某一类应用的计算设备。一般的设计方法是将一片或多片FPGA、CPU和存储器等组合到一起，FPGA作为协处理器加速程序中的一些可并行执行的部分，例如循环体等，同时由CPU管理FPGA的重构方式。这种结构的计算设备在当时被称为可重构计算机；这种计算方式也被称为可重构

计算。

从不同的研究角度出发，对可重构计算的理解也不尽相同。目前比较公认的定义是由加州大学伯克利分校可重构技术研究中心的 Andre Dehon 和 John Wawrzynek 于 1999 年 ACM 设计自动化国际会议上提出的一种广义的定义[4]。该定义将可重构计算视为一类新型的计算机组织结构，并具有区别于其他组织结构的突出特点：

(1) 区别于 ASIC 的制造后芯片可编程能力。

(2) 区别于通用微处理器能在很大程度上实现算法到计算引擎的空间映射。

从计算的本质来说，传统的计算模式可分为时间域上的计算和空间域上的计算。前者以基于微处理器的计算为代表，它指用于完成计算任务的计算资源具有一定的通用性，能够被计算任务中的各个操作分时复用；后者以基于 ASIC 的计算为代表，它指在计算资源集合中有专门的资源用于实现特定的计算任务。计算过程能够有效利用计算任务间的并行性以获得更高的数据吞吐量和更低的计算延时。

目前可重构计算以 FPGA 为技术基础，根据可重构器件配置文件中的编程信息，改变其中逻辑单元的功能以及互连方式，从而改变计算系统的功能。它是一种时空域上的计算模式，既能在设计实现时利用可重构器件定制专用的计算部件，又能够对计算资源进行复用以实现多个不同的计算任务。如图 6-1 所示，可重构计算弥补了微处理器实现和 ASIC 实现之间的性能差距，计算速度与 ASIC 相当，同时具备类似微处理器实现的灵活性。可重构计算为更广泛的应用提供了灵活有效的计算解决方案。

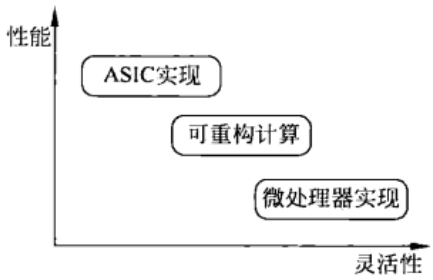


图 6-1　三种实现方式比较

如果进一步比较 ASIC 专用芯片、通用型微处理器和可重构计算的实现方式，可以发现在它们之间还有一些介于单纯的时间和空间之间的计算方式。我们可以按照确定计算任务的最后时间进行区分，这个时间称为计算任务或计算功能的绑定时间，即在任务绑定之后，芯片上的任务功能就不能被修改。如表 6-1 所示，对于专用芯片来说，当芯片完成设计时其功能就已经确定了，之后的流片、测试和销售都不能改变芯片的功能。因此 ASIC 芯片的功能绑定时间为“设计时”。对于掩膜可编程门阵列(MPGA)或者结构化 ASIC 芯片来说，它的底层晶体管或逻辑单元等连接是共享的，只有上面数层用于用户电路的定制。也就是说，对于 MPGA 或者结构化 ASIC 来说，当芯片完成制造时，所有版图信息及芯片功能就已经确定，因此它们的绑定时间是“流片时”。对于 FPGA 或者通用型的微处理器来说，在芯片制造和销售后，用户可以通过编程和编译产生可编程文件，不管是适合于 FPGA 的位流文件还是通用处理器的可执行文件。当这些文件在编译时产生二进制文件后，任务的功能一般不可能在之后的下载或者运行时发生变化，因此计算任务的绑定时间是编译时。对于可重构计算来说，当二进制文件被下载到 FPGA 后，还可以通过任务管理软件，在合适的时候自动对 FPGA 本身进行动态局部配置下载。具体下载时的任务需要根据运行时的环境来决定，即用户不能事先决定在一个确定的时刻究竟哪些任务被下载到 FPGA 执行，因此基于 FPGA 的可重构计算的任务绑定时间是“运行时”。由于单存的通用型处理器在运行时可执行程序本身是只读，不能对可执行程序本身进行动态更改，因此经典的软件程序并不具

有可重构计算的优势。

表 6-1 各种芯片的计算任务绑定时间

计算任务绑定时间	设计时	流片时	编译时	运行时
ASIC 芯片	√			
MPGA、结构化 ASIC 等		√		
静态配置 FPGA 和通用型处理器			√	
可重构系统				√

6.2 可重构系统及其分类

FPGA 作为目前最常用的可重构逻辑器件，是可重构计算的实现基础。以下所说的可重构系统就是指基于 FPGA 的可重构系统，它可简单定义为至少包含一个可重构硬件模块的计算系统。其中硬件模块的功能可以被最终用户修改。修改过程主要通过对系统中的 FPGA 进行全部重构或部分重构来实现。该系统可以在只增加少量硬件资源的情况下，将软件实现和硬件实现的优点合二为一。它的出现使传统意义上硬件和软件的界限变得模糊，硬件系统得以软件化或虚拟化[5]；或者说软件任务可以硬件化，用于提高计算效率。

可重构系统从被提出开始，就在一些数据密集型应用领域显示出了强大的计算性能和数据处理能力，尤其擅长于算法内部蕴涵很大并行性和流水性的应用。其中最具代表性的是美国超级计算机研究中心研制的 SPLASH 2。在基因组分析的应用中，它比当时的 SPARC 10 工作站的运算速度整整快了 2500 倍；若做灰度图像的中值滤波器，则比 SPARC 10 快了几乎 140 倍[6,7]。SPLASH 2 使用了 17 块 FPGA 而 SPARC 10 只有 1 个 CPU。按芯片数平均，在以上两个应用中，每块 FPGA 分别使性能提高了 147 倍和 8 倍。这一结果充分显示了可重构系统的应用潜力，引起了学术界和工业界对可重构系统的极大兴趣。

目前常见的可重构系统多包含 CPU 和以 FPGA 为代表的可重构器件，或者由其演变扩充的器件。可重构器件中包含一定数量的可重构处理单元(reconfigurable processing units，RPUs)。以下将按照耦合方式、单元粒度和重构方式对可重构系统结构进行分类介绍。

6.2.1 系统耦合方式

Todman 等在文献[8]中根据 CPU 与可重构器件之间的耦合方式将现有的可重构系统结构分为以下五类：

(1) 可重构处理单元作为外部独立的处理单元，如图 6-2 所示。

这是最为松散的耦合方式，可重构处理单元通过 I/O 接口(如网络、总线或者背板等)与 CPU 相连，通常用于 CPU 和可重构器件之间通信较少的场合。SPLASH 是采用这种结构的典型系统。这类系统的典型优点是设计简单、编程方便，并且可以根据需要灵活地选用不同的宿主机和 CPU 结构，但由于所有的数据交换都需要通过外部的接口总线来完成，这部分的性能成为瓶颈，从而限制了应用的性能提升。

(2) 可重构处理单元作为附加的计算模块，如图 6-3 所示。

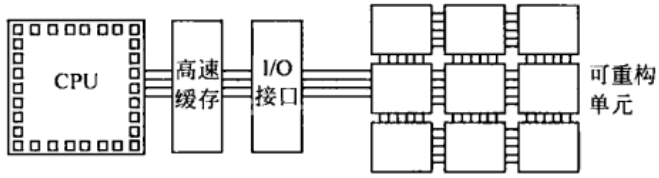


图 6-2　可重构处理单元作为外部独立的处理单元

在这种方式下，CPU 的数据缓冲对于可重构处理单元是不可见的。可重构处理单元与 CPU 之间的通信多以标准的通信原语完成，它们之间的通信延时较大。RAW 系统采用该结构[9]。由于多处理机的结构类似，该结构可借鉴多处理器领域内的多种编程方法和模式。

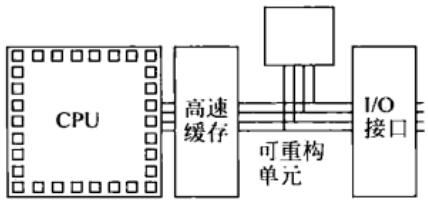


图 6-3　可重构处理单元作为附加的计算模块

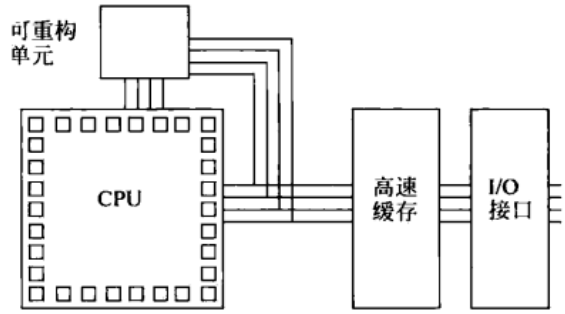


图 6-4　可重构处理单元作为 CPU 的协处理器

(3) 可重构处理单元作为 CPU 的协处理器，如图 6-4 所示。

在这种方式下，CPU 较少干预协处理器的计算任务处理。通常情况下，CPU 只是在初始化可重构处理单元后为其准备必要的数据；协处理器独立完成计算任务后将结果返回至 CPU。与前一种方式相比，协处理器能够以较小的延时与 CPU 和主存进行通信。Chameleon 为基于该结构的典型系统[10]。

(4) 可重构处理单元嵌入 CPU 内部，如图 6-5 所示。

在这种方式下，可重构处理单元作为功能可变的执行单元，CPU 内的寄存器、流水线都对可重构处理单元可见。该方式允许在传统编程环境下通过定制指令执行计算任务，适合于发掘程序内部的指令级并行，典型系统如 GARP[11]。

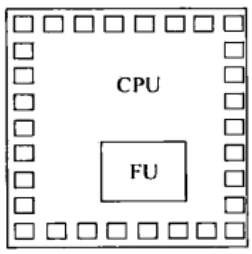


图 6-5　可重构处理单元嵌入 CPU 内部

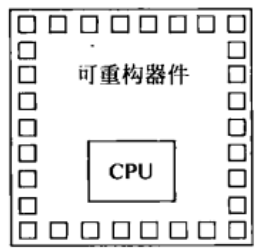


图 6-6　处理器嵌入到可重构器件内

(5) CPU 被嵌入到可重构器件中，如图 6-6 所示。

这是伴随着可重构器件技术进步而出现的新结构。被嵌入的微处理器可以是硬核 CPU 或者软核 CPU，可重构逻辑器件与通用处理器的整合减轻了通信和访存开销，同时降

低了可重构系统设计的复杂度。采用这种方式的典型可重构器件如 Xilinx Virtex 系列 FPGA，其中可按照需要选择内嵌 PowerPC 硬核处理器[12]或 Microblaze 软核处理器[13]。

上述五种结构各有利弊。可重构处理单元与 CPU 之间的耦合度越高，它们之间的通信代价越低。但是 CPU 与可重构处理单元之间的交互也越频繁，否则可重构处理单元独立运行的时间将会很短。更松散的耦合形式允许可重构处理单元与 CPU 之间有更大的并行性，但需要付出更大的通信代价。

6.2.2 可重构单元粒度

根据可重构单元粒度，可将可重构系统分为细粒度(fine grained)、粗粒度(coarse grained)和混合粒度(hybrid grained)三类。

细粒度重构是指每一位信号线有各自独立的可重构结构，是位级(bit-level)的可重构技术。粗粒度结构主要用于字宽数据通道的实现。由于设计时针对操作进行了优化，因此它在以字宽为单位的操作上，能获得比细粒度结构的系统更快的速度，同时也比细粒度的组合更节省芯片面积。但由于粗粒度结构无法按操作数的位宽变化进行优化，因此在执行按位操作或字长变化操作时，效率比细粒度结构差。如第 2 章的 2.4 节所示，当前新型的系统级 FPGA 器件均嵌入了一些粗粒度的功能块，例如存储器、DSP 模块甚至微处理器。这种结构可以称为混合粒度。当然，还有一种混合粒度结构的实现方式是把 FPGA 作为 IP 核嵌入到一个 SoC 芯片内部，从而使得 SoC 具备一定的灵活性。

6.2.3 系统重构方式

基于常规 SRAM 结构的 FPGA，通常只能用于实现系统的静态重构，其原因在于 FPGA 所实现的功能取决于下载到 SRAM 中的全部配置数据。在配置过程中，系统的逻辑功能在时间轴上发生断裂，无法实现动态接续。根据目前商用 FPGA 容量及配置方式的不同，其完全配置时间达到毫秒级，乃至秒级。

支持动态重构的 FPGA 具有特殊的 SRAM 结构。各 SRAM 单元能被独立地访问配置，且它们的功能互不影响。通过读取不同位置上的配置数据，可以直接实现器件内部逻辑单元的重构。以具备动态重构能力的 Xilinx Virtex Ⅱ系列 FPGA 为例，其配置 SRAM 以垂直帧的形式划分排列。每一帧的宽度为 1 位，长度从顶部边沿到底部边沿。而对于同样支持器件动态重构的 Xilinx Virtex-4 系列，每一帧的长度不再从上到下横跨整个器件，而是只跨越了 16 个 CLB。所有的配置操作均以“帧”为单位，配置帧的长度依赖于器件的容量大小。

现有的可重构器件主要支持以下几种重构方式。

1. 单上下文

在单上下文(single context)方式下，每次重写配置信息都必须重写整个配置存储器，如图 6-7 所示。

在这种方式下，即使仅需要修改其中的一部分配置信息，也必须将可重构器件停下来进行配置。尽管该方式有上述缺陷，但由于具有最为简单的配置控制电路，目前仍被广泛应用。支持单上下文方式的器件以 Altera 公司的 FPGA 为代表。

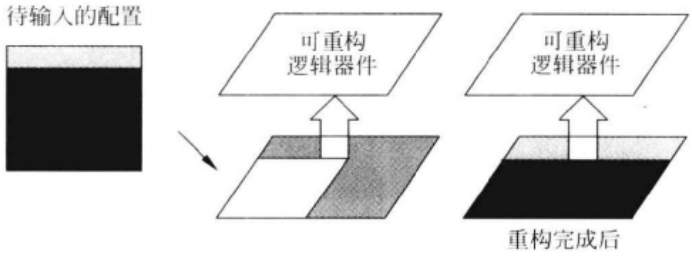


图 6-7　单上下文重构方式

2. 多上下文

采用多上下文(multi-context)配置方式的可重构器件最早出现在文献[14]中，其主要思路是为器件的逻辑阵列准备多个配置文件，存储在不同的地址中，如图 6-8 所示。

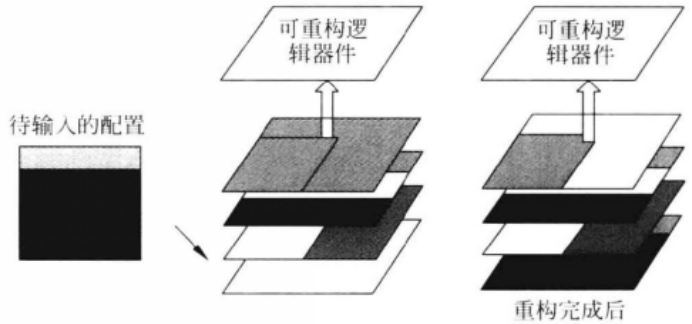


图 6-8　多上下文重构方式

其中某个配置文件可以按照计算任务的需求转换为有效状态，并在很短的时间内用该配置文件对可重构器件进行配置。与单上下文方式相比，多上下文方式的优点在于可以更快地切换配置文件，配置切换时间为纳秒级。而采用单上下文方式的器件重构时间为毫秒级。此外，在多上下文方式中，未激活的上下文可以随时被重写而不干扰系统的运行。

由于配置存储器会占据大量片上存储空间，所以这类器件的逻辑资源规模受到很大限制。目前针对这一类型的可重构器件的研发还处于试验性阶段[15]，尚缺少真正实用、商业化的大规模器件支持。

3. 动态部分重构

动态部分重构(dynamic partial reconfiguration)在每次重构时只需要有选择地对可重构器件上的部分资源进行重新配置，而不会影响到器件上的其他资源，如图 6-9 所示。

动态全局重构在每次重构时需要重构整片器件，这将中止计算任务的执行，直接影响了计算任务的执行连贯性。而利用动态部分重构技术，能够在不影响器件上其他硬件电路正常运行的前提下，对特定的可重构逻辑资源区域进行重构。同时，由于每次重构只需要对器件上的部分资源进行重新配置，减少了配置文件大小，从而缩短了系统重构时间。目前，Atmel 和 Xilinx 提供了具备动态部分重构能力的商用 FPGA。

根据可重构器件所支持的上述重构方式，可重构系统可分为静态可重构系统和动态可重构系统两类[2]。其中，静态可重构系统也称编译时重构系统，是指系统在开始执行计算任

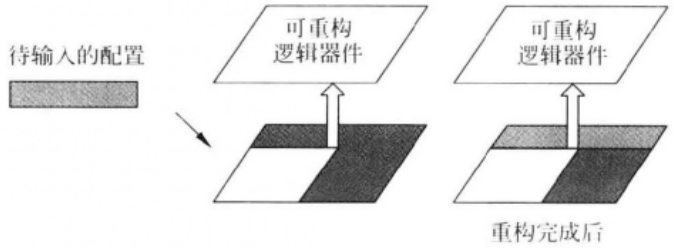


图 6-9　动态部分重构方式

务之前被配置到可重构器件上，并且在整个系统的运行过程中可重构器件上的配置将保持不变，直至完成该计算任务，如图 6-10(a)所示。静态可重构是目前商用 FPGA 所广泛支持的重构方式。

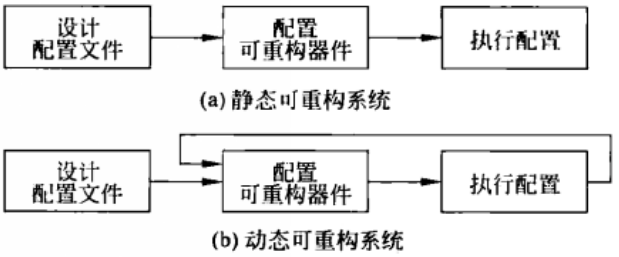


图 6-10　静态可重构系统和动态可重构系统

动态重构系统又称运行时重构系统，如图 6-10(b)所示，它能够在计算任务运行的同时对可重构器件上的逻辑资源进行重构。计算任务被划分为多个配置文件，每次在可重构器件上加载的配置文件与计算任务中的一部分相对应，因此在计算任务的执行过程中需要对可重构器件进行多次重构。

按照配置粒度，动态可重构系统可进一步细分为动态全局重构系统和动态部分重构系统(以下简称部分可重构系统)。动态全局重构系统在每次重构时都需要重新配置可重构器件上的全部计算资源，它在设计时将计算任务在时间维度上划分为多个阶段，每个阶段对应一个占用全部可重构逻辑资源的器件配置文件，在运行时将每个阶段对应的配置文件按照时间顺序先后配置到可重构器件上。与动态全局重构不同，动态部分重构可在运行时对可重构器件中的部分资源进行重构，而不会影响到器件上的其他资源。它可以按照应用需求分时复用、实时重构器件上的逻辑资源。对计算任务不仅要在时间域上进行划分，而且要在空间域上进行划分，在重构时只需要对任务所占用的资源进行重构。

表 6-2 为三种重构方式之间的执行特征比较。

表 6-2　重构方式执行特征比较

重构方式	资源绑定方式	任务配置方式
静态重构	任务设计时	系统运行前
动态全局重构	任务设计时	系统运行时
动态部分重构	任务运行时	系统运行时

与静态重构相比，动态重构能够在系统运行过程中按照计算任务的需要将相应阶段的配置文件加载到器件上，能够更及时地满足计算任务需求的变化。动态部分重构能够在系

统运行过程中，根据当前可重构器件上的资源使用情况，为新的计算任务分配合适的资源空间，完成计算任务与可重构逻辑资源的运行时绑定；而在静态重构和全局动态重构技术中，计算任务均在运行前被绑定到整片可重构逻辑器件上。

6.3 模块化的部分可重构系统设计方法

6.3.1 设计方法

动态部分重构允许对可重构系统的一部分进行重新配置，配置过程中其余部分的工作不受影响，可实现对系统资源的时分复用并有效缩短重构开销，是当前可重构计算领域的主要发展趋势之一。

Xilinx 支持两种部分可重构系统设计方法：基于差异(difference-based)的设计和基于模块(module-based)的设计[16]。基于差异的设计方法多用于两个差别较小的设计之间，只需改动查找表功能、改写 BRAM 内容或 I/O 引脚等属性，设计人员可使用 FPGA Editor 工具对经过布局布线后的电路描述文件(native circuit description，NCD)进行修改。由于该设计方法只适用于对设计进行小的改动，因此应用范围有限。

模块化的设计方法是一种粗粒度的、以用户硬件模块为基本重构单位的设计方法。该方法将应用设计划分为两类模块：固定模块和可重构模块，然后分别对每个模块进行设计、综合，并对模块的布局以及所占据资源的大小进行约束。最后再将所有模块的实现结果进行有机组合以完成整个设计。在重构过程中，固定模块不做任何变动，而具有多个设计版本的可重构模块则可按照应用需求进行重构，并有效地减少重构过程中的配置数据量。图 6-11 为基于 Xilinx Virtex-Ⅱ Pro FPGA 的部分可重构系统布局示意图。

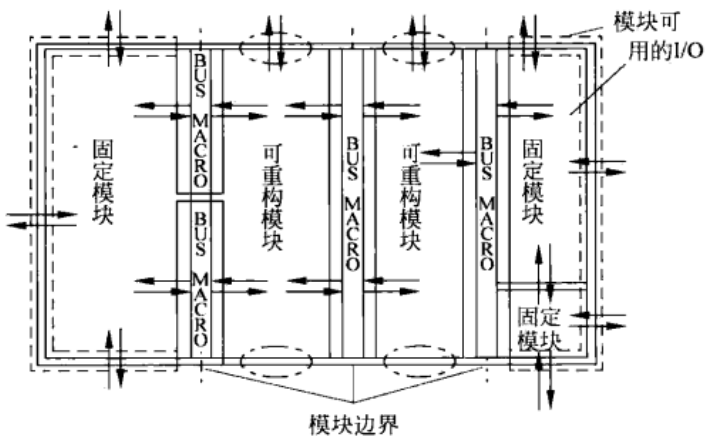


图 6-11 基于 Xilinx Virtex-Ⅱ Pro FPGA 的部分可重构系统布局示意图[16]

6.3.2 设计流程

基于模块的部分可重构系统设计流程主要包括两个阶段：第一阶段为顶层设计/综合；第二阶段为模块设计实现。图 6-12 为具体设计流程描述，其中“顶层设计/综合”、“模块设计/综合”属于第一阶段；“初始资源预估”、“模块实现”和“模块集成”组成模块设计实现阶段。最终生成整个系统实现的位流文件以及可重构部分对应的位流文件。

1. 顶层设计/综合

系统设计人员根据系统需求，创建顶层设计文件并对其进行综合。设计人员在顶层设计文件中需要定义全局信号和各模块的 I/O 接口，并实现模块之间的连接，以保证它们之间能够正常通信。

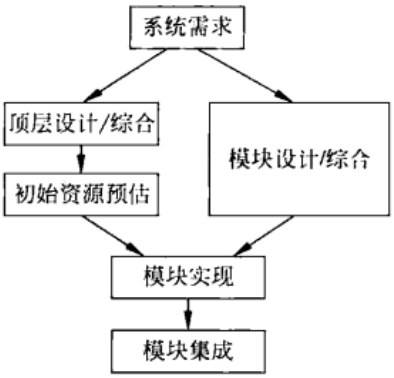


图 6-12 模块化的部分可重构系统设计流程

2. 模块设计/综合

在模块设计文件中设计人员需要描述模块的具体功能实现。所有模块必须使用全局时钟；模块间不能直接共享除全局时钟、全局复位信号、全局使能以外的任何信号。每个模块的 I/O 接口必须与顶层设计中的接口定义一致。

3. 初始资源预估

在完成上述顶层设计/综合、模块设计/综合后，需要对模块占用的资源进行预先估计，并根据估计结果创建顶层设计约束文件。在初始资源预估阶段需要完成以下步骤：

(1) 放置全局逻辑；

(2) 在目标芯片上定义模块的放置位置以及模块所占据资源的大小，即对可重构模块所在的可重构区域和其他非重构模块所在的静态区域进行布局；

(3) 定义每个模块输入输出端口的位置；

(4) 定义约束文件，包含所有模块的布局和时序约束。

4. 模块实现

在可重构系统设计过程中，需要将每个模块单独实现。首先利用综合工具将模块设计描述综合为网表，而后经过转换、映射、布局布线得到其在 FPGA 内的位流。每个模块进行布局布线时受到顶层约束的限制，只能添加模块内部的时序约束，不允许对顶层的约束进行改动。模块实现时只考虑模块内部的连接关系以及约束文件中规定的总线宏位置，不受其他模块的影响。

5. 模块集成

在模块集成阶段，在上一阶段得到的模块布局布线结果都将会被保留，以保证每个模块的性能。将各模块实现根据顶层设计合并起来形成一个完整的可重构系统设计，生成完整的位流文件和用于实现模块动态重构的部分位流文件。在系统运行前，首先将完整的位流文件下载到 FPGA，并保证可重构系统开始正常运行。在系统运行过程中，用户可以根据应用需求将对应的部分位流文件下载到 FPGA 进行动态重构。

6.4 可重构系统设计实例

根据上述模块化设计方法，本节以实时音频信号处理为例，利用包括 Xilinx ISE 8.2.1、EDK 8.2 以及 PlanAhead 8.2.8 在内的 EAPR(early access partial reconfiguration)部分可

重构系统设计工具集[17]，在 Xilinx XUP 开发板上实现一个包含高通、低通以及带通滤波功能在内的可重构音频信号处理系统。

1. 系统结构及模块划分

如图 6-13 所示，可重构音频信号处理系统由支持部分重构的 XC2VP30 FPGA 及相关外设组成。在 FPGA 内包含 PPC 405 硬核 CPU 和部分可重构模块(partial reconfigurable module，PRM)，两者共享 OPB 总线。CPU 时钟频率和总线频率均为 100MHz。系统可通过配置控制器 ICAP(internal configuration access port)[18] 对 FPGA 进行运行时重构以加载用户所需的可重构模块，实现对音频信号的高通/低通/带通滤波处理，而对其他正在运行的部分没有影响。

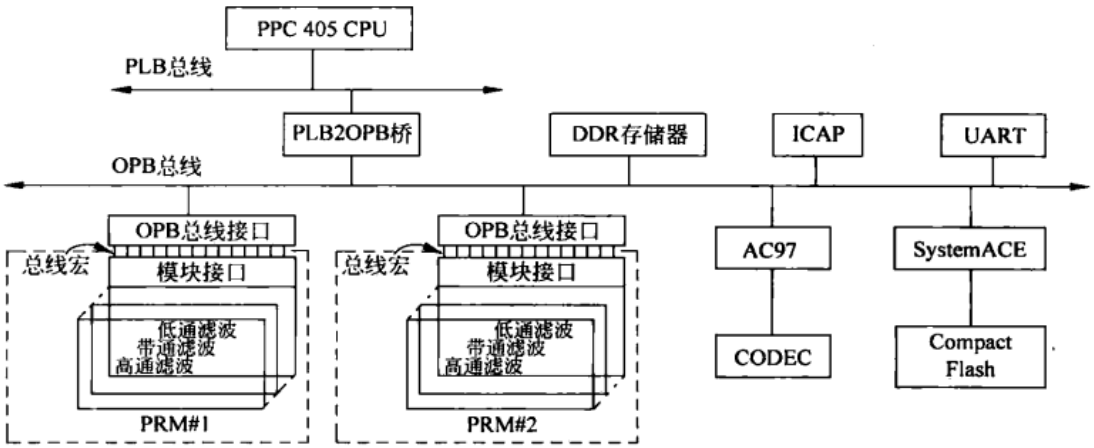


图 6-13　可重构音频信号处理系统

按照 EAPR 可重构系统设计方法，平台在设计时被划分为用于左、右两个声道音频信号的可重构处理模块，即图中的“PRM ＃1”和“PRM ＃2”，以及除 PRM 以外的基本设计两部分。它们分别位于 FPGA 部分可重构区域(partial reconfigurable region，PRR)PRR_0、PRR_1 以及静态区域。PRR 和静态区域之间通过总线宏[17]通信。“PRM ＃1”或“PRM ＃2”在运行时可以动态切换，实现“高通滤波”、“带通滤波”和“低通滤波”等功能。这就好像在同一块可编程硬件资源上可以虚拟化不同的任务，大大提高硬件资源的利用率。图 6-14 为系统布局和资源分配情况。

以部分可重构区域 PRR_0 为例，在如图 6-15 所示的用户约束文件中对系统布局、模块分配和总线宏放置作如下描述，并通过设计规则检查以确保正确的设计布局。

2. 模块设计与实现

系统中音频信号采样频率为 48000Hz。以高通滤波处理可重构模块为例，其设计参数及对应的模块 VHDL 语言描述分别为：

```
High Pass Filter:                    fpass-10000Hz;
                                     fstop-5500Hz;
                                     Astop-80db;
```

Apass-1 db

```
component highpass_10k  is
        port (
              ce_1: in std_logic;
              clk_1: in std_logic;
              sample_in: in std_logic_vector(15 downto 0);
              sample_valid: in std_logic;
              sample_out: out std_logic_vector(15 downto 0);
              sample_out_valid: out std_logic
        );
end component;
```

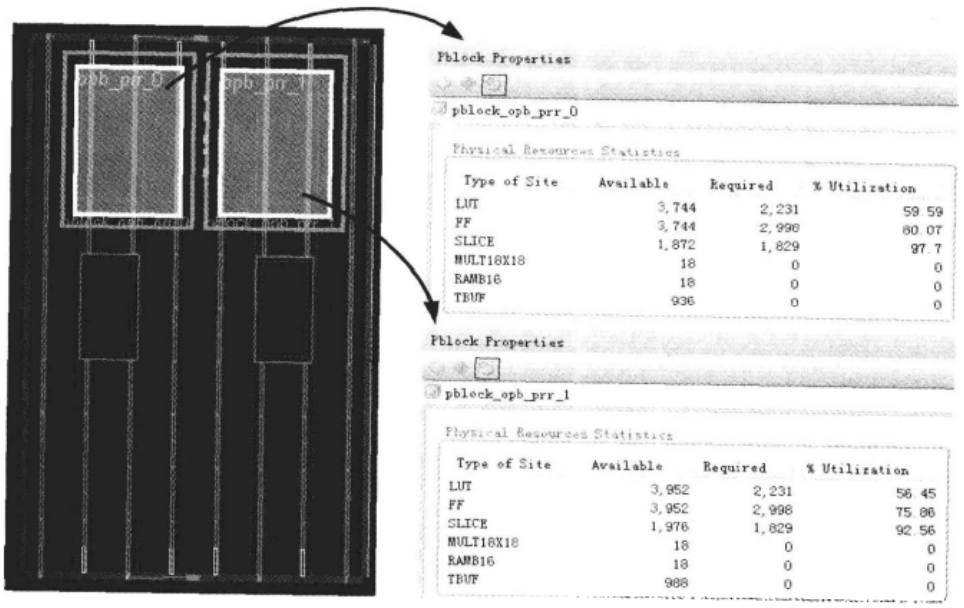


图 6-14 系统布局和资源分配

```
## 部分可重构区域 PRR_0 布局约束
AREA_GROUP "pblock_opb_prr_0"
RANGE = SLICE_X8Y104:SLICE_X15Y155, SLICE_X16Y104:SLICE_X27Y155, SLICE_X28Y104:
SLICE_X39Y155, SLICE_X40Y104:SLICE_X43Y155;
...
AREA_GROUP "pblock_opb_prr_0"
RANGE = RAMB16_X1Y13:RAMB16_X1Y18, RAMB16_X2Y13:RAMB16_X2Y18, RAMB16_X3Y13:RAMB16
_X3Y18;
AREA_GROUP "pblock_opb_prr_0" MODE = RECONFIG;
## 可重构模块分配
INST "opb_prr_0" AREA_GROUP = "pblock_opb_prr_0";
## 总线宏放置
INST "opb_socket_bridge_0/opb_socket_bridge_0/ABUS_BM_GENERATE[1].ABUS_BM" LOC =
SLICE_X6Y120;
INST "opb_socket_bridge_0/opb_socket_bridge_0/Control1_BM"
LOC = SLICE_X6Y142;
```

图 6-15 部分可重构模块对应的用户约束文件示例

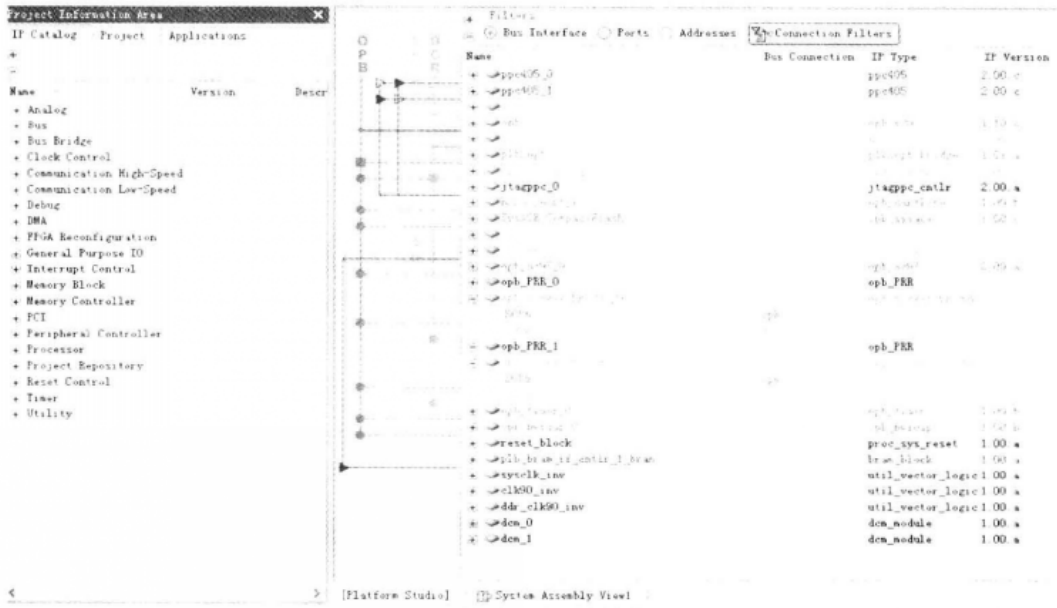
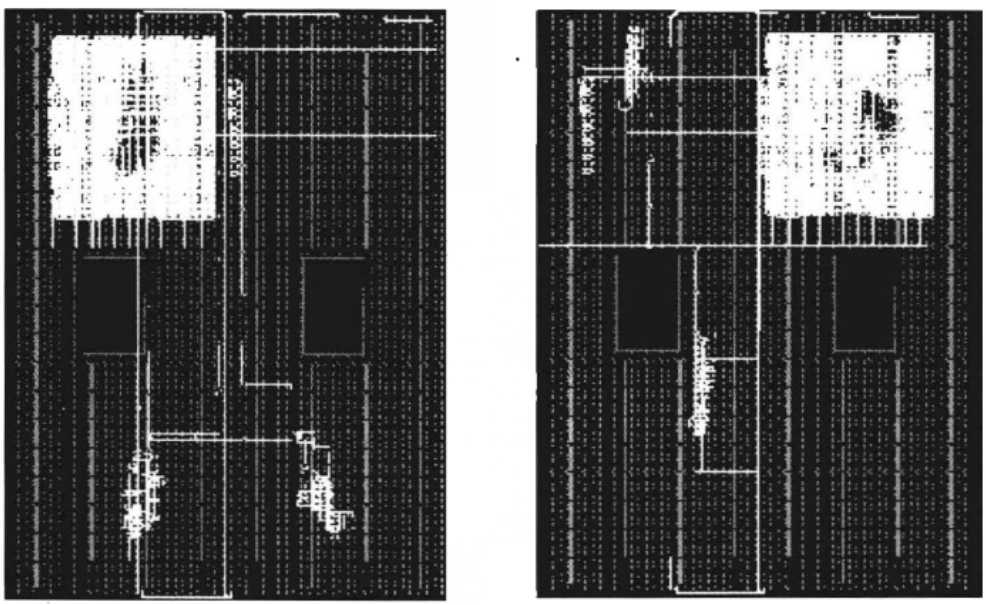


图 6-16　封装为 OPB IP 核的 PRM

如图 6-16 所示，两个音频处理模块首先被单独综合为网表，并通过总线接口封装为 OPB 总线上的 IP 核：opb_PRR_0 和 opb_PRR_1，其对应的总线接口分别为 opb_socket_bridge_0 和 opb_socket_bridge_1。封装后的各 PRM 经过综合、映射、布局布线后，各自生成部分位流文件，分别用于左、右两声道音频信号的实时滤波处理。图 6-17 为两个高通滤波处理 PRM 对应的布局布线结果。



(a) PRM#1布局布线结果　　(b) PRM#2布局布线结果

图 6-17　两个高通滤波处理 PRM 的布局布线结果

3. 模块集成

在最终的模块集成阶段，综合工具将系统中的各个模块进行合并，以形成完整的 FPGA 配置文件供初始下载使用。

4. 系统功能验证

图 6-18 为可重构实时音频信号处理系统运行时配置情况。将系统中各可重构模块视为可配置的硬件任务，以部分位流文件形式存在，对应的硬件任务库位于 Compact Flash 存贮器中。在运行时，用户可根据需要从硬件任务库中读取对应的硬件任务，分配到可重构处理单元上执行。图中选择了对左声道音频信号进行带通滤波处理，其部分位流文件大小为 383008B，所需的配置时间为 27781370ns(OPB_Timer 使用的计数时钟频率为 100MHz)，即 27.78ms；与软件实现的带通滤波器相比，处理时间仅为软件实现的 23%。

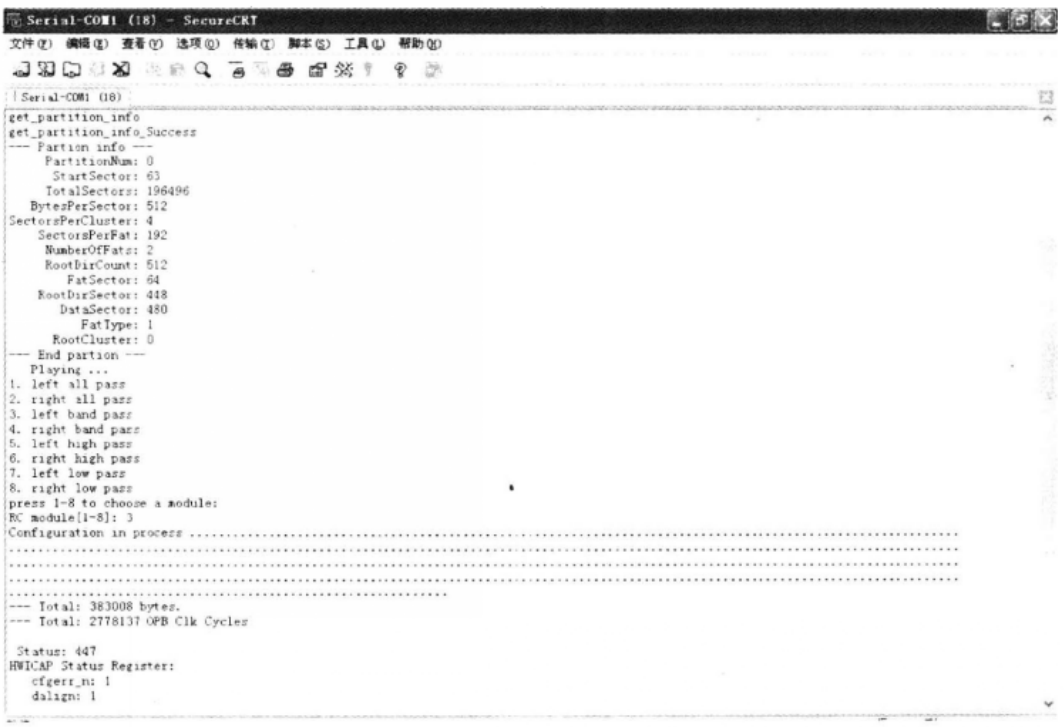


图 6-18 可重构实时音频信号处理系统运行时配置

从上述设计实例可知：利用模块化设计方法实现的部分可重构系统，其硬件任务重构时间可达到毫秒量级，可重构处理方式在探索空间复用实现灵活性的同时能够有效地兼顾硬件的效率。

6.5 本章小结

可重构系统作为新兴的计算平台，可以通过硬件可编程来适应数据密集型、计算密集型应用的需求。它的主要目标是希望通过硬件可编程实现自适应计算，获得灵活高效的处理

能力。目前随着可重构系统复杂度的不断增加，越来越需要考虑从系统级进行设计，以提高设计效率。此外在运行环境支持方面，如何通过操作系统屏蔽系统底层实现细节，有效管理可重构计算单元以提高资源利用率，也是可重构系统研究中的热点问题之一。

习题

1. 以 Xilinx XC2VP30 FPGA 为例，简要说明其配置帧结构。

2. FPGA 的配置过程是在所选择的配置模式下，将位流文件下载到 FPGA 的过程。配置过程主要包括四个步骤：(1)清除配置内存；(2)初始化；(3)装载配置流数据；(4)启动设备。试描述 FPGA 部分可重构配置流程并加以说明。

3. 配置重定位(relocation)技术能够有效地提高可重构系统的资源利用率，PARBIT 是实现配置重定位功能的一个软件工具[19]，试说明其配置重定位实现机制。

4. 如何有效地缩短系统的重构时间、提高可重构计算系统性能是可重构计算系统研究中的一个关键问题。给出至少四种缩短配置时间的方法并加以分析。

参考文献

[1] Compton K. Reconfigurable Computing: A Survey of Systems and Software. ACM Computing Surveys, 2002, 34(2): 171-210

[2] Kiran Bondalapati, Viktor K Prasanna. Reconfigurable Computing Systems. Proceedings of the IEEE, 2002, 90(7): 1201-1217

[3] Gerald Estrin. Reconfigurable Computer Origins: The UCLA Fixed-Plus-Variable (F+V) Structure Computer. IEEE Annals of the History of Computing, 2002, 24(4): 3-9

[4] Dehon A, Wawrzynek J. Reconfigurable Computing: What, Why, and Implications for Design Automation, Proc. 36th ACM/IEEE Conf. on Design Automation, ACM Press, 1999: 610-615

[5] Villasenor J, Mangione-Smith W H. Configurable Computing, Scientific American, 1997, 11: 23-27

[6] Arnold J M. The Splash 2 software environment. IEEE Workshop on FPGAs for Custom Computing Machines, 1993: 88-93

[7] Arnold J M, Buell D A, Hoang D T, et al. The Splash 2 Processor and Applications. IEEE International Conference on Computer Design: VLSI in Computers and Processors, 1993: 482-485

[8] Todman T J, Constantinides G A, Wilton S J E, et al. Reconfigurable Computing: Architectures and Design Methods. IEEE Proceedings on Computers and Digital Techniques, 2005: 152(2): 193-207

[9] Waingold E, Taylor M, Srikrishna D, et al. Baring it All to Software: Raw Machines. Computer, 1997, 30(9): 86-93

[10] I. Chameleon Systems. CS2000 Advance Product Specification, Chameleon Systems, 2000

[11] Callahan T J, Hauser J R, Wawrzynek J. The Garp Architecture and C Compiler. Computer, 2000, 33(4): 86-69

[12] Xilinx Inc. Virtex Ⅱ Datasheet, http://www.xilinx.com/

[13] Xilinx Inc. Microblaze Processor Reference Guide, Xilinx Documentation, 2005

[14] DeHon A. DPGA-Coupled Microprocessors: Commodity ICs for the Early 21st Century. IEEE Workshop on FPGAs for Custom Computing Machines, 1994: 31-39

[15] Hariyama M, Ogata S, Kameyama M. A Multi-Context FPGA Using Floating-Gate-MOS

Functional Pass-Gates. IEICE Transactions on Electronics, E89-C(11),2006: 655-1661

[16] Xilinx Inc. Two Flows for Partial Reconfiguration: Module Based or Difference Based, http://www.xilinx.com/bvdocs/appnotes/xapp290.pdf

[17] Xilinx Inc. Early Access Partial Reconfiguration User Guide, Xilinx User Guide UG208, Version 1.1, March 6, http://www.xilinx.com/, 2006

[18] Xilinx Inc. OPB HWICAP (v1.00.b), Xilinx Document DS280, 2005

[19] Horta E L, Lockwood J W, Kofuji S. Using PARBIT to Implement Partial Run-time Reconfigurable Systems. Field-Programmable Logic and Applications: Reconfigurable Computing Is Going Mainstream, 2002: 93-101

第 7 章

系统级 FPGA 综合设计实例

前面各章已经介绍了系统级 FPGA 在嵌入式系统中的应用设计技术。本章通过一个具体的应用实例，全面地展示基于 FPGA 的嵌入式系统的设计过程。该实例采用 Altera 的 DE2 开发板实现一个数字录音机，具有录音、放音、快进、快退等基本功能。

7.1 DE2 开发板简介

DE2(Development and Education board II)是 Altera 公司针对大学及研究机构推出的一款多媒体开发与教学板，如图 7-1 所示，它是学习数字逻辑、FPGA 和计算机组成的实验平台。DE2 为用户提供了丰富的外设及多媒体特性，其中所提供的材料中包含了大部分设计的源代码。它不仅适用于大专院校实验室的教学研究与课题制作，也适合工业界作为复杂数字系统的开发平台。

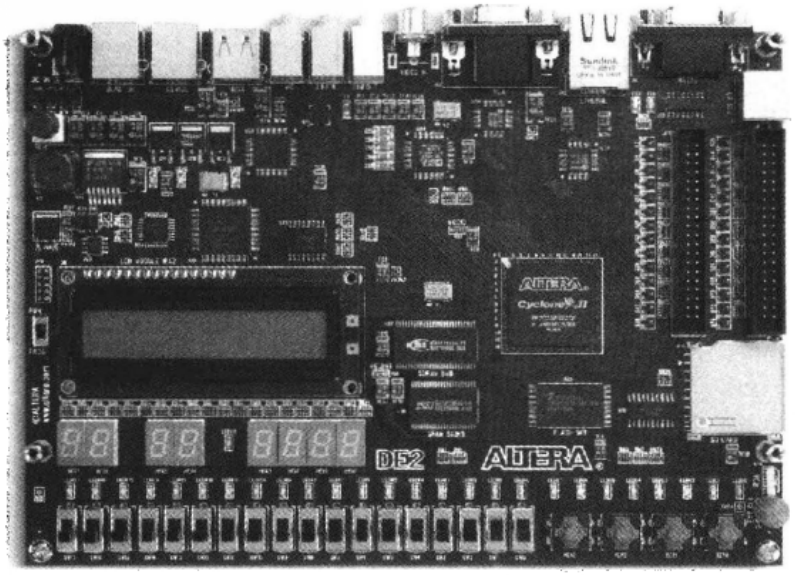


图 7-1 Altera 公司提供的 DE2 开发板

DE2 提供了以下硬件资源。

- Altera Cyclone Ⅱ系列的 EP2C35F672C6 FPGA 器件。
- 主动串行配置器件 EPCS16。

- 板上内置用于编程调试和用户 API 设计的 USB Blaster 插口。
- 两个板上时钟源，分别为 50MHz 和 27MHz。
- 12KB SRAM，8MB SDRAM，1MB Flash，SD 卡接口。
- 4 个按键，18 个拨动开关。
- 9 个绿色 LED 灯，18 个红色 LED 灯，8 个七段数码管。
- 2×16 字符的 LCD 模块。
- 24 位 CD 品质的音频编/解码器，带有麦克风输入、线路输入和线路输出插口。
- VGA DAC 及 VGA 输出接口。
- 支持 NTSC 和 PAL 制式的 TV 解码器及 TV 接口。
- 10M/100M 以太网控制器及网络接口。
- USB 主从控制器及接口。
- RS-232 收发器及 9 针连接器。
- PS2 鼠标/键盘连接器。
- IRDA 收发器。
- 带二极管保护的两个 40 脚扩展接口。

DE2 的硬件框图如图 7-2 所示，所有的外部器件都能连接到 FPGA 芯片。这为 DE2 的使用提供了极大的灵活性，用户可以通过 FPGA 的配置实现任意的系统设计。

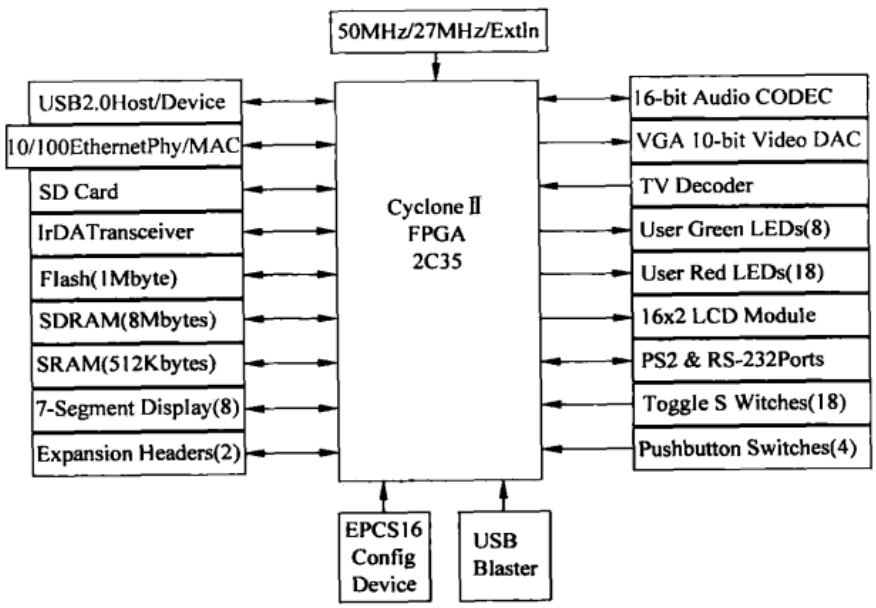


图 7-2 DE2 开发板的硬件框图

7.2 应用实例硬件设计

7.2.1 系统架构设计

DE2 开发板提供了足够的硬件资源用以实现录音机的各项功能。首先，DE2 具有音频编/解码器，可实现音频信号的输入输出；使用按键和拨动开关实现录音机的操作功能；使

用 LCD、七段数码管和 LED 灯显示状态信息；使用 SDRAM 存放录下的声音数据；最后，使用 FPGA 搭建一个基于 Nios Ⅱ的 SOPC 系统实现录音机的核心控制功能。其系统框图如图 7-3 所示。

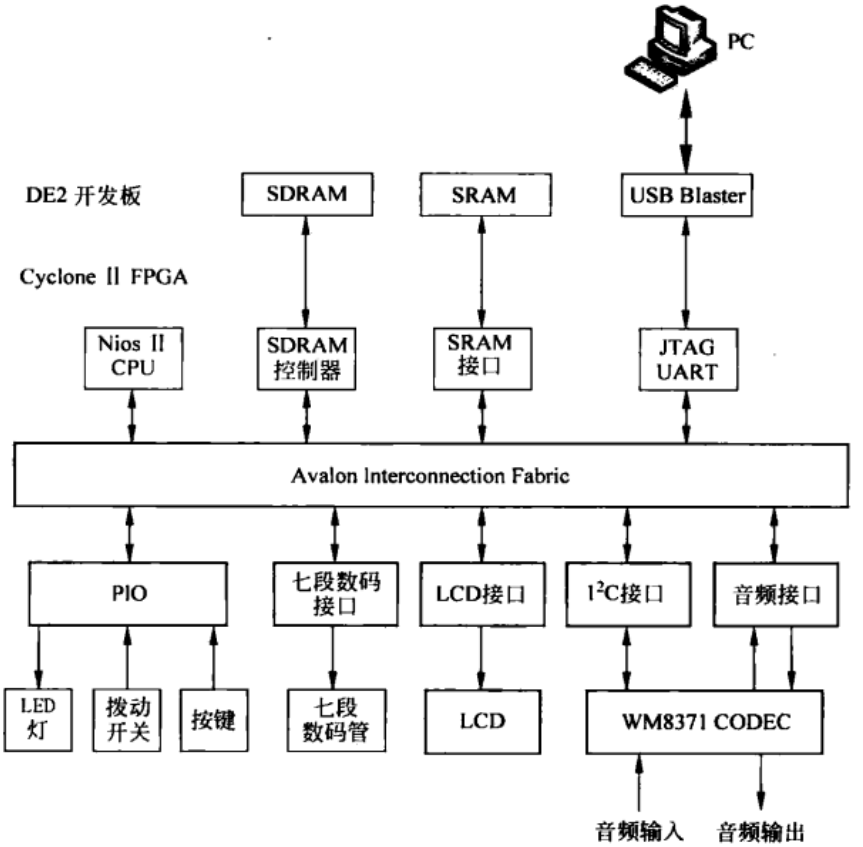


图 7-3　数字录音机系统框图

应用实例采用 Altera SOPC Builder 工具来实现，表 7-1 列出了 SOPC 系统中包含的模块。录音机实例中用到的许多外设不在 SOPC Builder 提供的 IP 库中，需要自行设计。本章 7.3 节会详细介绍这些外设 IP 组件的设计。

表 7-1　SOPC 系统模块

模块名称	说　明	主要参数配置
cpu	Nios Ⅱ处理器	标准配置
tristate_bridge	三态总线桥，用来连接外部存储器芯片（如 SRAM 和 Flash 等）	
sram	异步静态 RAM，自定义 IP 组件，用来放置录音机程序	容量：512KB 数据宽度：16b
sdram	同步动态 RAM 控制器，用来放置音频数据	数据宽度 16b，4 个 bank，12 行 8 列，容量 8MB

续表

模块名称	说　明	主要参数配置
pll	锁相环，为 SDRAM 和音频器件提供时钟信号：c0 提供给 SDRAM，c1 提供给音频器件	输入时钟：50MHz c0 输出：50MHz，−3ns 相移 c1 输出：18.432MHz
sysid	系统 ID，用来检查软硬件是否匹配	
sys_clk_timer	系统时钟定时器	周期：1ms
i2c	I^2C 接口，自定义 IP 组件	
audio	音频输入/输出，自定义 IP 组件	
pio_ledg	绿色 LED 灯	9 位输出
pio_ledr	红色 LED 灯	18 位输出
pio_key	按键（只用 3 个按键，另一个用作系统复位）	3 位输入
pio_sw	拨动开关	18 位输入
seg7	8 位七段数码管，自定义 IP 组件	
lcd	字符 LCD	
jtag_uart	JTAG 串口，用于软件调试	

在 SOPC Builder 的界面中可配置这些模块的参数以及模块间的连接，如图 7-4 所示。由于 SRAM 用作程序存储器，CPU 的指令总线通过三态总线桥连接到 SRAM 存储器，数据总线连接其他外设。

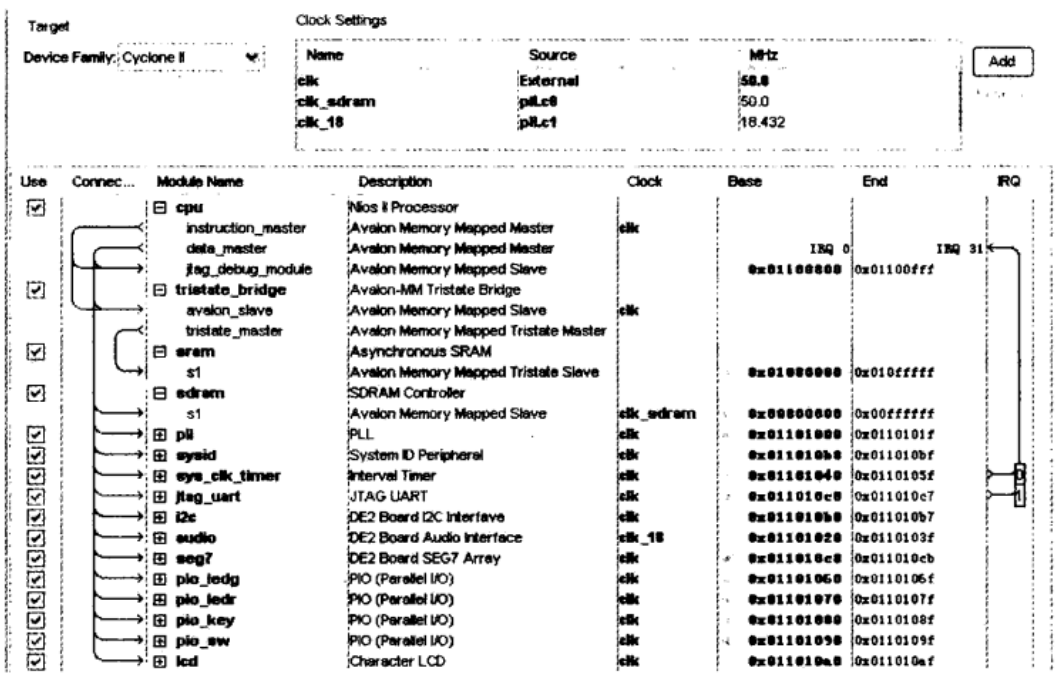


图 7-4　SOPC Builder 设计界面

7.2.2　顶层模块实现

顶层模块在 Quartus II 工具中实现，其中包含 SOPC Builder 生成的 SOPC 系统模块及

其 IO 引脚连接关系。数字录音机实例采用了原理图格式实现顶层模块，能直观地显示系统的顶层接口，如图 7-5 所示。注意 SRAM 存储器地址线的特殊连接，三态总线桥输出的是字节地址，而 SRAM 存储器要求的是 16 位字地址，因此在连接时要错开一位。

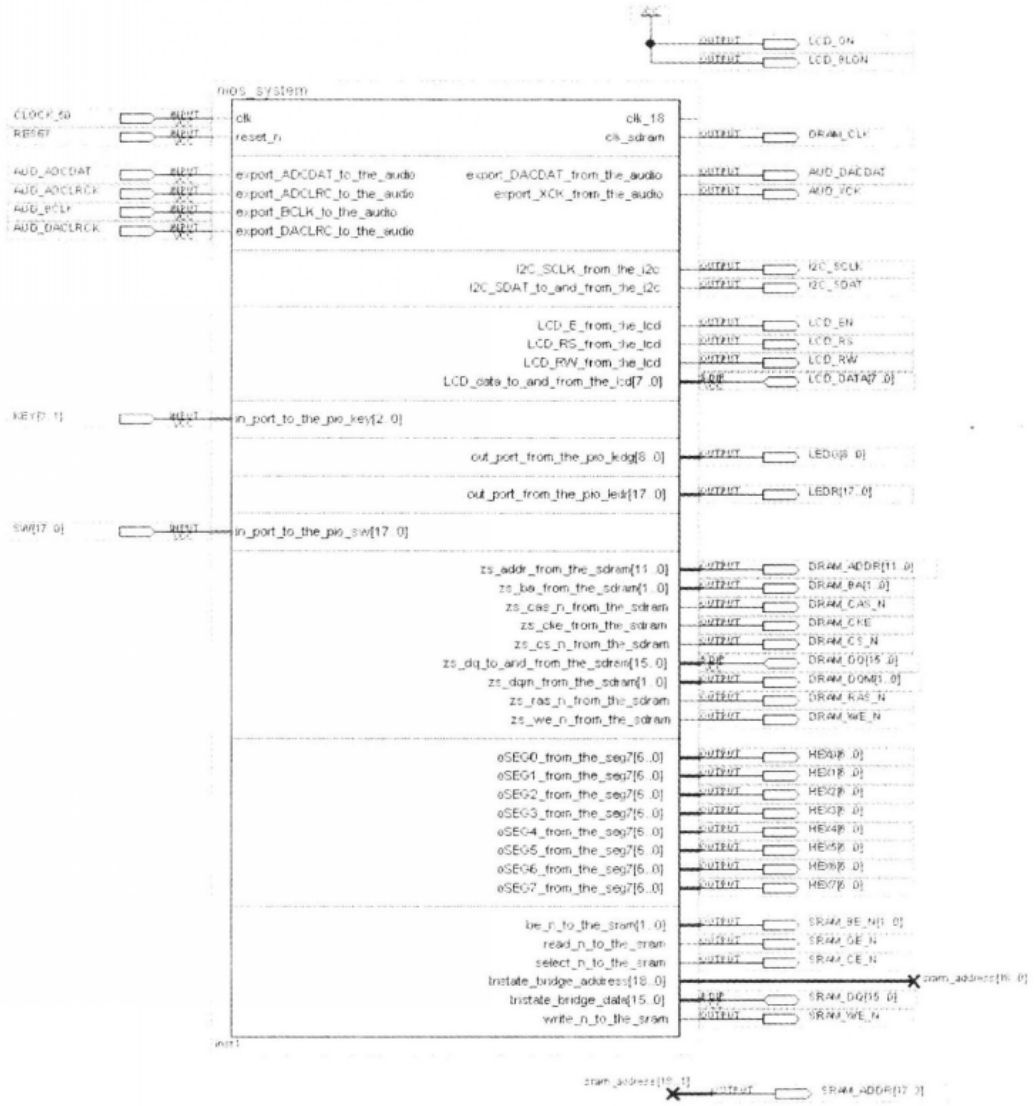


图 7-5　录音机系统顶层模块

7.3　自定义外设及其驱动程序设计

DE2 开发板的光盘提供了一批设计实例，在这些设计实例中包含了板上各种外设的接口电路。但这些接口电路大都没有按照 SOPC Builder 工具的 IP 组件规范来实现，也没有符合 HAL 规范的设备驱动程序。笔者在设计数字录音机实例时重新整理了实例需要的外设接口，增加了必要的总线接口电路，并配上了设备驱动程序，得到了四个自定义外设 IP 组件。下面分别介绍这些组件的设计内容。

7.3.1 SRAM 接口组件

SOPC Builder 只针对某些开发板提供了 SRAM 的接口组件，并未提供一个通用的 SRAM 接口组件。但我们可以对 SOPC Builder 提供的 SRAM 接口组件进行修改，以匹配 DE2 板上的 SRAM 器件。我们选用了“IDT71V416 SRAM”组件，它已提供了可配置的容量参数，但数据宽度固定为 32 位，不可配置。通过修改组件的 class. ptf 文件，将数据宽度变成用户界面上可配置的参数，就使这个 SRAM 接口组件变成了一个相对通用的组件，能够与 DE2 板上的 SRAM 器件相匹配。

值得注意的是，我们用来修改的基础组件是基于 PTF 文件的老版本组件，其设计文件均为文本格式，容易修改；而新版本的 SRAM 组件是用 Java 实现的，且未提供源代码，难以做进一步的修改。

7.3.2 七段数码显示组件

DE2 开发板有 8 个七段数码管，可显示 8 位数字。七段数码组件提供了方便的 API 接口，用户只需提供待显示数字，组件便能以十六进制或十进制的方式显示。

在设计组件时需要考虑软硬件功能的划分，将一位数字值转换为数码管的控制信号是一个查表操作，适合于硬件实现；而计算十进制数的各位数字涉及乘除法操作，用软件实现比较方便。

1. 硬件设计

七段数码组件的硬件结构如图 7-6 所示。它主要由译码电路和总线接口电路两部分组成。组件从总线接收一个 32 位数字，存放在寄存器中，然后再通过译码，输出到 8 个七段数码管。组件共有 8 个译码单元，每个译码单元将 4 位二进制数转换为七段数码管的控制信号。

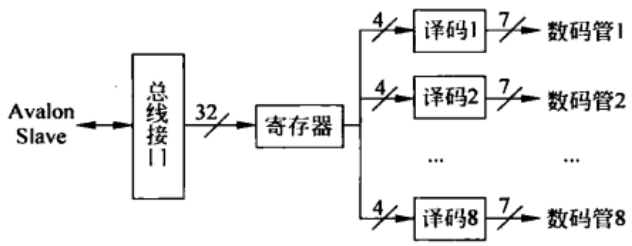


图 7-6 七段数码组件硬件结构图

图 7-7 是七段数码组件顶层模块的 Verilog 语言描述。Avalon 总线是同步总线，设备需要在时钟上升沿检查总线控制信号，并做出相应处理。七段数码组件的总线接口比较简单，只需在写使能有效时将总线数据写入寄存器即可，Avalon 总线不要求设备产生应答信号，在组件描述文件中定义固定的等待周期即可。

在硬件设计完成后，使用 SOPC Builder 中的组件编辑器(component editor)建立该组件的描述文件。组件最重要的配置信息是其接口信号的定义，如表 7-2 所示。七段数码组件只是一个简单的只写组件，因此其总线接口只需包含写数据和写使能信号，不需要地址、

```
module SEG7_LUT_8 (
    oSEG0, oSEG1, oSEG2, oSEG3, oSEG4, oSEG5, oSEG6, oSEG7,
    iDIG, iWR, iCLK, iRST_N );
input    [31:0]    iDIG;                  // 待输出的数字
input    iWR, iCLK, iRST_N;               // 总线控制信号
output   [6:0]     oSEG0,... ,oSEG7;      // 数码管控制信号
reg      [31:0]    rDIG;

// 总线接口
always@(posedge iCLK or negedge iRST_N)
begin
    if(!iRST_N)
        rDIG <= 0;
    else
    begin
        if(iWR)
            rDIG <= iDIG;
    end
end

// 译码电路
SEG7_LUT  u0  (oSEG0, rDIG[3:0]);
...
SEG7_LUT  u7  (oSEG7, rDIG[31:28]);

endmodule
```

图 7-7　七段数码组件顶层模块描述

读数据和读使能等信号。conduit 接口用来连接组件自定义的外部信号，在七段数码组件中用来定义连到七段数码管的输出信号。Avalon 总线接口的等待状态也是一个值得关注的组件参数，对于大多数用户自定义组件，采用默认设置（读等待 1 周期，写等待 0 周期）即可。

表 7-2　七段数码组件接口信号定义

名　　称	接口类型	信号类型	方向	宽度/b
iDIG	avalon_slave	writedata	input	32
iWR	avalon_slave	write	input	1
iCLK	clock_reset	clk	input	1
iRST_N	clock_reset	reset_n	input	1
oSEG0 .. oSEG7	conduit	export	output	7

2. 设备驱动程序设计

按照 HAL 规范的要求，驱动程序的头文件和 C 源代码分别放在 IP 组件的 HAL/inc 以及 HAL/src 目录下。头文件中定义了设备模块的实例化和初始化的宏，以及 API 函数原型。C 代码包含了 API 函数的实现。头文件必须命名为“<组件名>.h”，两个宏也必须命名为“<大写组件名>_ INSTANCE”和“<大写组件名>_ INIT”，这样才能被 Nios Ⅱ IDE 正确使用。

七段数码组件包含两个 API 函数，SEG7_Hex 用来显示十六进制数据，而 SEG7_Decimal 用来显示十进制数据。组件不需要明确的实例化和初始化，只需将它们定义为空操作，如图 7-8 所示。

```
// Macros used by alt_sys_init
#define FD_SEG7_INSTANCE(name, device) extern int alt_no_storage
#define FD_SEG7_INIT(name, device) while (0)

// Prototypes
void SEG7_Hex(alt_u32 base, alt_u32 data);
void SEG7_Decimal(alt_u32 base, alt_u32 data);
```

图 7-8 七段数码组头文件片段

7.3.3 I²C 接口组件

I²C(inter-integrated circuit)总线是一种两线式串行总线，通常用于连接微控制器及其外围设备。DE2 开发板采用 I²C 总线配置视频和音频编解码芯片。

I²C 接口组件的硬件结构如图 7-9 所示，它由 Avalon 总线接口、FIFO 和 I²C 控制器组成。FIFO 用来实现 Avalon 总线和 I²C 总线的速度匹配，I²C 控制器将数据串行输出，并接收应答信号。本组件只实现了 I²C 的输出功能，未实现 I²C 的输入，因此不能用来读出音/视频芯片的控制寄存器。

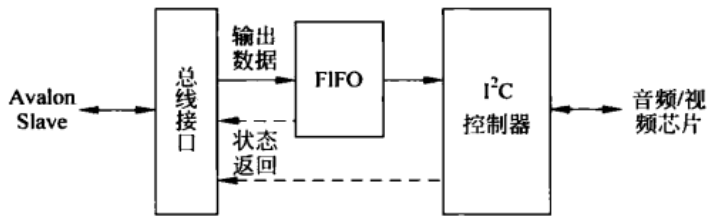


图 7-9 I²C 接口组件硬件结构图

I²C 接口组件的总线接口代码如图 7-10 所示。它定义了两个寄存器，地址 0 为数据寄存器，用来写入 I²C 输出数据；地址 1 为状态/控制寄存器，写入该寄存器时用来复位 FIFO，读出该寄存器能得到 FIFO 和 I²C 控制器的状态。组件中的寄存器并不一定要在 HDL 代码中用寄存器来实现。在 I²C 接口组件数据寄存器就是 FIFO，状态/控制寄存器使用组合电路实现。

组件中的 FIFO 模块是用 Quartus II MegaWizard Plug-in Manager 生成的，主要参数为：宽度 24 位，深度 64 字，读/写时钟分离，异步复位。

本节重点介绍 IP 组件基于 Avalon 总线接口的实现，有兴趣的读者可参考文献[1]来了解 I²C 控制器的实现。如表 7-3 所示，组件的接口信号定义如下：总线接口包括了地址线、读写数据和读写使能信号，两个寄存器只需 1 位地址就能区分；Conduit 信号是 I²C 的时钟和数据线，连接外部的音频和视频芯片。

```
'define DATA_ADDR    0
'define CMD_ADDR        1
////////// fifo clear
always @ (posedge avs_clk)
begin
    if (avs_reset)
        fifo_clear <= 1'b0;
    else if (avs_write && (avs_address == 'CMD_ADDR))
        fifo_clear <= avs_writedata[0];
    else if (fifo_clear)
        fifo_clear <= 1'b0;
end
////////// write i2c data to fifo
always @ (posedge avs_clk)
begin
    if (avs_reset || fifo_clear)
    begin
        fifo_write <= 1'b0;
    end
    else if (avs_write && (avs_address == 'DATA_ADDR))
    begin
        fifo_writedata <= avs_writedata;
        fifo_write <= 1'b1;
    end
    else
        fifo_write <= 1'b0;
end
////////// response data to avalon, fifo & i2c status
assign avs_readdata = {29'b0, i2c_error, fifo_full, fifo_empty};
```

图 7-10　I^2C 组件的总线接口代码

表 7-3　I^2C 组件接口信号定义

名称	接口类型	信号类型	方向	宽度/b
avs_clk	clock_reset	clk	input	1
avs_reset	clock_reset	reset	input	1
avs_address	avalon_slave	address	input	1
avs_read	avalon_slave	read	input	1
avs_readdata	avalon_slave	readdata	output	32
avs_write	avalon_slave	write	input	1
avs_writedata	avalon_slave	writedata	input	32
I2C_SCLK	conduit	export	output	1
I2C_SDAT	conduit	export	input	1

I^2C 接口组件的驱动程序仅包含一个头文件，定义了组件的寄存器及其读写宏，I^2C 接口组件头文件如图 7-11 所示。

```
#define I2C_FIFO_FULL(base) \
    ((IORD(base, I2C_STATUS_PORT) & MASK_STATUS_FIFO_FULL) != 0)
#define I2C_FIFO_EMPTY(base) \
   ((IORD(base, I2C_STATUS_PORT) & MASK_STATUS_FIFO_EMPTY) != 0)
#define I2C_ERROR(base) \
   ((IORD(base, I2C_STATUS_PORT) & MASK_STATUS_I2C_ERROR) != 0)
#define I2C_WRITE(base, dev, ctrl, data) \
  IOWR(base, I2C_FIFO_PORT, (((dev) << 16) | ((ctrl) << 9) | (data)))
```

图 7-11 I²C 接口组件头文件

7.3.4 音频输入/输出接口组件

DE2 的音频输入/输出是由 Wolfson 公司的低功耗立体声 24 位音频编/解码芯片 WM8731 实现。WM8731 包含了线路输入、麦克风输入及耳机输出，具有 8～96kHz 可调的采样频率，16～32 位可调的数据宽度。

1. 硬件设计

WM8731 通过三组信号连接到 FPGA：I²C 配置、ADC 输入和 DAC 输出，如图 7-12 所示。WM8731 芯片有主模式和从模式两种工作模式。在主模式下由编/解码芯片本身产生 ADC 和 DAC 的时钟信号，控制数据传输率；而在从模式下，则是由外部（通常是微处理器）向编/解码芯片提供 ADC 和 DAC 的时钟信号。本组件要求将编/解码芯片设置为主工作模式，16 位数据宽度。

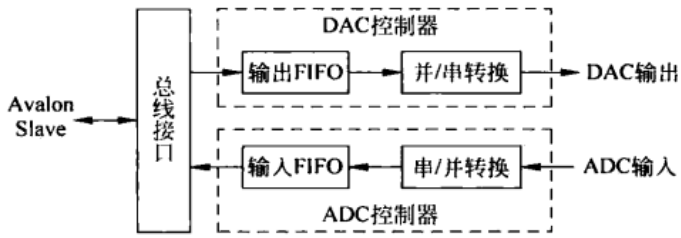


图 7-12 音频输入/输出组件硬件结构图

与 I²C 接口类似，ADC 和 DAC 接口也是串行的，其数据传输速率与采样频率的设置相关，因此也需要用 FIFO 来解决它们和 Avalon 总线间的速度差异。音频输入/输出组件由总线接口、ADC 控制器和 DAC 控制器等模块组成，其中 ADC 控制器包含 FIFO 和串/并转换电路，DAC 控制器包含 FIFO 和并/串转换电路。组件的两个 FIFO 以及总线数据宽度均为 32 位，高 16 位和低 16 位分别包含左声道和右声道的数据。FIFO 也是用 Quartus 工具中的 MegaWizard Plug-in Manager 生成的。主要的参数为：宽度 32 位，深度 256 字，读/写时钟分离，异步复位。

组件的总线接口代码如图 7-13 所示。它定义了两个寄存器：地址 0 为数据寄存器，用来读写音频数据；地址 1 为状态/控制寄存器，读出该寄存器时能得到 FIFO 的状态，写入该寄存器时用来使 FIFO 复位。

```
'define DAC_FIFO_ADDR     0
'define ADC_FIFO_ADDR     0
'define CMD_ADDR          1
'define STATUS_ADDR       1
////////// fifo clear
always @ (posedge avs_clk)
begin
    if (avs_reset)
       fifo_clear <= 1'b0;
    else if (avs_write && (avs_address == 'CMD_ADDR))
       fifo_clear <= avs_writedata[0];
    else if (fifo_clear)
       fifo_clear <= 1'b0;
end
////////// write audio data(left&right) to dac - fifo
always @ (posedge avs_clk)
begin
    if (avs_reset || fifo_clear)
    begin
        dacfifo_write <= 1'b0;
    end
    else if (avs_write && (avs_address == 'DAC_FIFO_ADDR))
    begin
        dacfifo_writedata <= avs_writedata;
        dacfifo_write <= 1'b1;
    end
    else
        dacfifo_write <= 1'b0;
end
////////// read audio data from adc fifo
always @ (negedge avs_clk)
begin
    if (avs_reset)
    begin
        adcfifo_read <= 1'b0;
        data32_from_adcfifo <= 0;
    end
    else if ((avs_address == 'ADC_FIFO_ADDR) &
             avs_read & ~adcfifo_empty)
    begin
        adcfifo_read <= 1'b1;
    end
    else if (adcfifo_read)
    begin
        data32_from_adcfifo = adcfifo_readdata;
        adcfifo_read <= 1'b0;
    end
end
////////// response data to avalon
assign    avs_readdata = (avs_address == 'STATUS_ADDR) ?
                          {adcfifo_empty, dacfifo_full} :
                          data32_from_adcfifo;
```

图 7-13　音频输入/输出组件总线接口代码片段

音频组件的接口信号定义如表 7-4 所示。其总线接口和 I²C 组件相同，Conduit 信号连接到音频芯片。

表 7-4 音频组件接口信号定义

名称	接口类型	信号类型	方向	宽度/b
avs_clk	clock_reset	clk	input	1
avs_reset	clock_reset	reset	input	1
avs_address	avalon_slave	address	input	1
avs_read	avalon_slave	read	input	1
avs_readdata	avalon_slave	readdata	output	32
avs_write	avalon_slave	write	input	1
avs_writedata	avalon_slave	writedata	input	32
export_DACLRC	conduit	export	input	1
export_DACDAT	conduit	export	output	1
export_ADCLRC	conduit	export	input	1
export_ADCDAT	conduit	export	input	1
export_XCK	conduit	export	output	1
export_BCLK	conduit	export	input	1

2. 设备驱动程序设计

音频输入/输出组件的设备驱动程序包含两部分内容：音频编解码芯片的配置和音频数据的输入/输出。音频芯片的配置是通过 I²C 实现的，因此驱动程序需要保存 I²C 模块的基地址。由于 I²C 接口组件未实现输入功能，驱动程序还需要维护音频芯片各个控制寄存器的值。为此，驱动程序定义了音频设备的数据结构，存放音频设备和 I²C 设备的基地址，以及音频芯片的控制寄存器值。为了使应用程序能够访问到这一数据结构，驱动程序使用了 HAL 的注册机制，将设备数据结构的实例同设备名称关联起来，应用程序可以通过设备名称得到指向设备数据结构的指针。

驱动程序定义了一组音频芯片的配置函数，包括音频芯片初始化、音量控制、采样频率控制和输入源选择等。其中音频芯片初始化函数按照录音机系统的要求对芯片进行了配置，打开 ADC 和 DAC，并将麦克风作为输入源。

音频数据的输入/输出功能的实现比较简单，只需定义一些宏来读写音频输入/输出设备的数据和状态寄存器即可。这些宏分两级定义：第一级通过基地址访问设备，第二级通过设备指针来访问。

驱动程序的头文件如图 7-14 所示。设备初始化时调用了 HAL 的设备注册函数。音频芯片的初始化操作在这时调用比较合适，但音频芯片初始化函数需要一个额外的参数来提供 I²C 设备的基地址，在这里不能提供，因此音频芯片的初始化就需要由用户程序显式调用了。

```
/* 音频设备结构 */
typedef struct {
  alt_dev dev;                    // HAL 字符设备,用来实现注册机制
  unsigned int base;              // 音频设备基地址
  unsigned int i2c_base;          // I²C 设备基地址
  alt_u16 i2c_reg[NUM_I2C_REG];  // 音频芯片控制寄存器
} fd_audio_dev;

/* Prototypes */
// find device by name
fd_audio_dev* AUDIO_find_dev(const char* name);

// audio configure
void AUDIO_Init(fd_audio_dev* dev, unsigned int i2c_base);
void AUDIO_SetInputSource(fdu_audio_dev* dev, alt_u8 InputSource);
void AUDIO_SetSampleRate(fdu_audio_dev* dev, alt_u8 SampleRate);
  ...
void AUDIO_WaitConfigure(fd_audio_dev* dev); // 等待配置完成

// play & record
#define AUDIO_DacFifoFull(dev)   AUDIO_DAC_FULL(dev->base)
#define AUDIO_DacFifoWrite(dev, data) \
                                 AUDIO_DAC_WRITE(dev->base, data)
#define AUDIO_AdcFifoEmpty(dev) AUDIO_ADC_EMPTY(dev->base)
#define AUDIO_AdcFifoRead(dev)   AUDIO_ADC_READ(dev->base)
#define AUDIO_FifoClear(dev)     AUDIO_FIFO_CLEAR(dev->base)

/* Macros used by alt_sys_init.c */
// 设备实例化
#define FD_AUDIO_INSTANCE(name, dev)  \
static fd_audio_dev dev = {           \
  { ALT_LLIST_ENTRY, name##_NAME },  \
  ((void*)( name##_BASE))             \
}
// 设备初始化
#define FD_AUDIO_INIT(name, d) alt_dev_reg(&d.dev)
```

图 7-14　音频输入/输出设备驱动头文件代码片段

7.4　软件设计

录音机系统的许多底层功能已经在设备驱动程序中实现了,这使得只需编写一个简单的顶层控制程序就可以实现录音机的各项功能。程序中定义了以下用户操作界面,如表 7-5 所示。

表 7-5　数字录音机的用户操作界面

接　口	功　能	接　口	功　能
KEY0	硬件复位	SW0	启动/停止
KEY1	快进	SW17	录音/放音
KEY2	快退	七段数码管	显示录/放音时间,精确到毫秒
KEY3	退回起始位置	LCD	状态显示

录入的声音存放在 SDRAM 存储器中。为节省存储空间，程序只保存一个声道的数据，在 48kHz 的采样频率下，8MB 的 SDRAM 可以记录 87s 的声音数据。

下面的代码是录音机系统的主程序，其中包括一个主循环。它根据当前系统状态执行相应的操作，检测按键和开关的状态，并作出响应。在录音状态下，不断从音频设备读出声音数据，将右声道数据存放到 SDRAM 中；在放音状态下，将 SDRAM 中单声道的数据复制到两个声道，输出到音频设备中。在 Nios Ⅱ IDE 软件属性设置中已将 LCD 设备制定为标准输出设备，因此主程序只需使用 printf 函数便可将字符串输出到 LCD 上，如图 7-15 所示。

```
// 使用 SDRAM 作为声音存储缓冲区，16 位数据宽度
#define BUF_SIZE (SDRAM_SPAN / 2)
alt_u16 * Buffer = (alt_u16 * )SDRAM_BASE;
alt_u32  Pos;                                    // 录/放音位置

fd_audio_dev * audio;                            // 音频设备
enum { IDLE, RECORD, PLAY, DONE } stat;          // 运行状态

int main() {
  audio = AUDIO_find_dev("/dev/audio");          // 获取设备指针
  AUDIO_Init(audio, I2C_BASE);                   // 初始化音频设备
  AUDIO_SetSampleRate(audio, RATE_ADC48K_DAC48K);

  Pos = 0; stat = IDLE;
  for (;;) {                                     // 主循环
    if (TestButton(REWIND_BUTTON))               // 退回起始位置
      Pos = 0;
    else if (TestButton(BACKWARD_BUTTON))        // 快退
      Pos = Pos < STEP ? 0 : Pos - STEP;
    else if (TestButton(FORWARD_BUTTON))         // 快进
      Pos = Pos >= BUF_SIZE - STEP ? BUF_SIZE - 1 : Pos + STEP;
    display_time();                              // 显示录/放音时间

    switch (stat) {
    case IDLE:
      if (TestSwitch(RUN_SW)) {
        if (TestSwitch(RECORD_SW)) {             // 启动录音
          printf("\nRecording...\n"); stat = RECORD;
        }
        else {                                   // 启动放音
          printf("\nPlaying...\n"); stat = PLAY;
        }
      }
      break;
    case RECORD:
      if (Pos >= BUF_SIZE || !TestSwitch(RUN_SW)) {
        printf("\nRecord finished\n"); stat = DONE;
      }
      else
        while (!AUDIO_AdcFifoEmpty(audio))       // 录音操作
```

图 7-15 主程序代码片段

```
            // 只保存一个声道
            Buffer[Pos++] = AUDIO_AdcFifoRead(audio) & 0xffff;
        break;
      case PLAY:
        if (Pos >= BUF_SIZE || !TestSwitch(RUN_SW)) {
          printf("\nPlay finished\n"); stat = DONE;
        }
        else
          while (!AUDIO_DacFifoFull(audio))          // 放音操作
            // 将一个声道的数据复制到另一个声道
            AUDIO_DacFifoWrite(audio, Buffer[Pos++] * 0x10001);
        break;
      case DONE:
        if (!TestSwitch(RUN_SW)) {
          printf("\nDigital Recorder\n"); stat = IDLE;
        }
        break;
      }
    }
    return 0;
}
```

图 7-15　（续）

作为书中的应用实例，本系统的功能设计得比较简单，但读者只需修改一下软件，就能实现许多额外的功能，例如输入源控制、采样频率设置、音量控制和 A-B 复读等。通过增加相应的硬件模块，还可以实现将录制的声音存入 SD 卡，以及播放 SD 卡中存放的声音文件。

参考文献

[1] 张志刚. FPGA 与 SOPC 设计教程——DE2 实践. 西安：西安电子科技大学出版社，2007

[2] Altera Corp. DE2 Development and Education Board User Manual，2006

[3] Altera Corp. Embedded Design Handbook，June，2008

[4] Altera Corp. Nios Ⅱ Software Developer's Handbook，May，2008

[5] Xilinx Inc. Embedded System Tools Reference Manual v9. 2i，September，2007

[6] Xilinx Inc. Device Driver Programmer Guide，June，2007

[7] David E Simon 著. 陈向群等译. 嵌入式系统软件教程. 北京：机械工业出版社，2005

附录 A

七段数码管显示设计实验

实验目的

(1) 了解基于 Verilog 语言设计数字电路的基本方法。

(2) 熟悉 FPGA 应用设计的基本流程。

(3) 掌握常用电路如分频器、译码器和小型数字电路的设计与优化。

实验内容

用 Altera 公司的 DE2 板上的 8 个七段数码管，实现一个符合以下要求的滚动显示电路模块。其顶层模块定义如图 A-1 所示。

```
module top (reset, clk, mode, char_idx, seg_idx, seg_out);
    input         reset;
    input         clk;
    input         mode;
    input         edit_ce;
    input[0:3]    char_idx;
    input[0:2]    seg_idx;
    output[0:55]  seg_out;      //8 displays with 7 inputs
endmodule
```

图 A-1　七段数码管显示的顶层模块电路接口

其中各个信号定义如下。

reset：全局异步复位信号，用 DE2 上的一个按钮实现，当按下按钮时 8 个数码管均显示为“0”。

clk：全局同步时钟输入信号。

mode：模式选择信号，用 DE2 上的一个开关实现。当 mode 为 0 时，模块处于内容编辑状态，用户可以设置 char_idx 来选择显示内容，设置 seg_idx 来选择当前编辑的数码管；当 mode 为 1 时，模块处于显示状态，8 个数码管的内容将循环滚动显示。

edit_ce：编辑使能信号，用一个开关实现。该信号用来防止 seg_idx 改变时意外改变其他数码管的内容。

char_idx：用于设置数码管的显示内容，用 DE2 上的 4 个开关实现。4'b0000 ～ 4'b1111 分别对应字符{'0', '1', '2', … '9', 'H', 'E', 'L', 'O', '空格'}。

seg_idx：用于指定当前所编辑的数码管，用 3 个开关实现。3'b000 ～ 3'b111 分别对应 DE2 上的 8 个数码管。

seg_out：数码管信号输出，连接到 DE2 上的 8 个七段数码管。

实验步骤

(1) 定义好整个电路的模块划分，包括模块的数目，各模块的接口与功能。

(2) 运行 Quartus，建立一个 Verilog/VHDL 设计项目，其中器件选择为 Altera Cyclone® Ⅱ EP2C35F672C6，如图 A-2 所示。

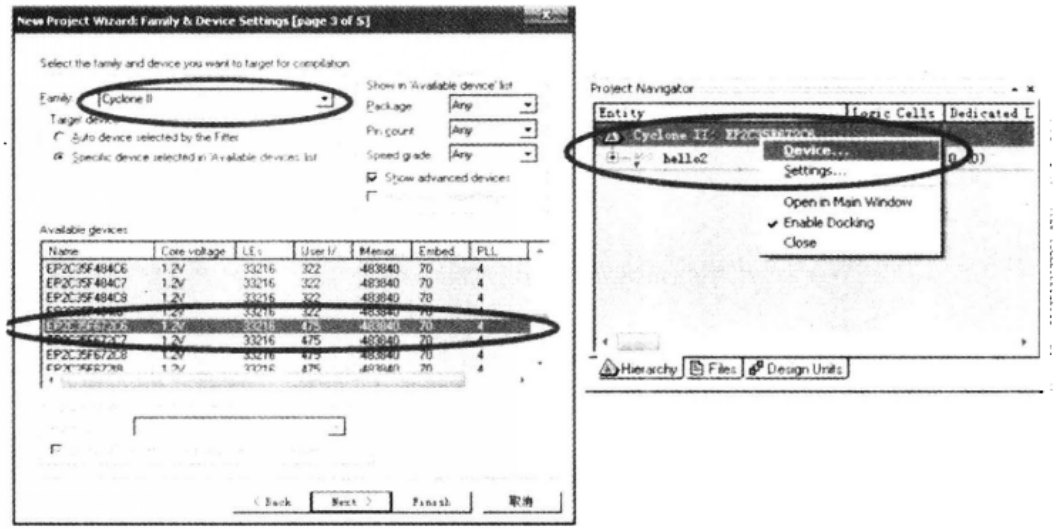


图 A-2　在 Quartus 中选择合适的 FPGA 器件

(3) 用硬件描述语言实现定义的各个模块，最后描述出顶层模块连接。

(4) 编写模块的仿真电路和激励，用 ModelSim 工具进行功能仿真，若仿真不通过则寻找原因并修改相应的模块。

(5) 利用 DE2 开发套件提供的 DE2_TOP. v 文件，将相应的开关管脚连接到顶层模块中，如图 A-3 所示。

(6) 导入 DE2 开发板提供的 DE2_pin_assignments. csv 文件，该文件为 DE2_TOP. v 的管脚约束文件，确保“top”模块的输入输出被锁定到正确的 FPGA 管脚上。

(7) 单击“编译”按钮，此时 Quartus 将执行综合、映射、布局布线等步骤并最终生成 FPGA 位流文件，如图 A-4 所示。编译结束后就会生成报告，显示模块的资源利用率、时序性能等信息。

(8) 启动下载程序将生成的位流文件下载到 FPGA 芯片上进行在线调试，如图 A-5 所示。若发现模块功能有误还可以利用 Quartus 工具进一步调试。

(9) 统计 Quartus 生成的资源利用率、时序性能和功耗分析等数据，进一步优化模块的设计。

```
module DE2_TOP
    (
        ////////////////////  Clock Input  ////////////////////
        CLOCK_27,                       //  27MHz
        CLOCK_50,                       //  50MHz
        .
        .
        .
    );

    .
    .
    .
top inst_top(
    .reset(KEY[0]),
    .clk(CLOCK_27),
    .mode(SW[0]),
    .edit_ce(SW[1]),
    .char_idx(SW[2:5]),
    .seg_idx(SW[6:8]),
    .seg_out({HEX0,HEX1,HEX2,HEX3,HEX4,HEX5,HEX6,HEX7})
    )
endmodule
```

图 A-3 顶层模块的连接关系

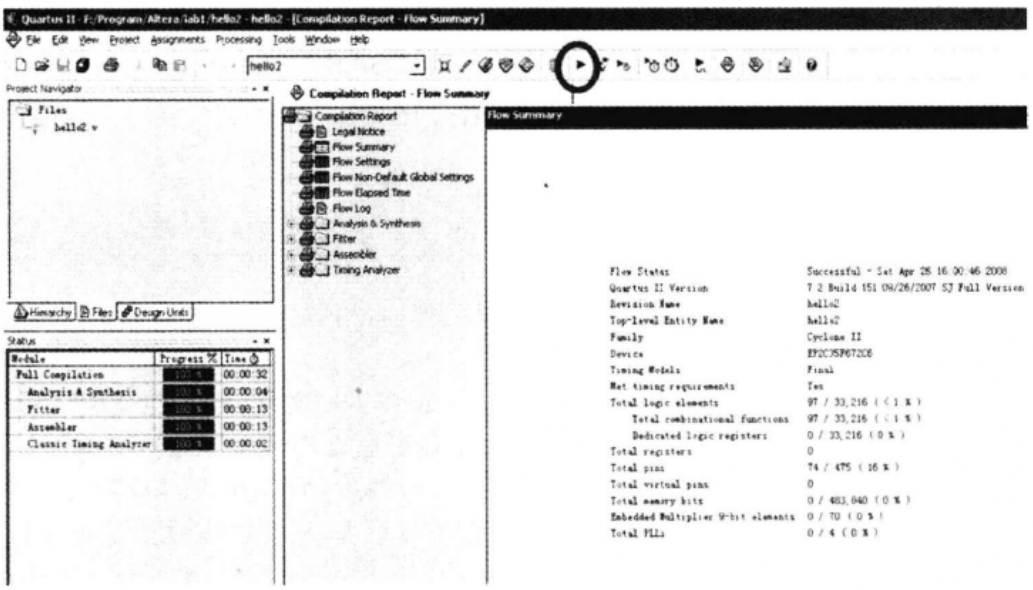


图 A-4 整体流程编译

实验小结

(1) 掌握用 FPGA 进行专用电路设计的流程和方法，理解 FPGA 实现应用电路的基本原理。

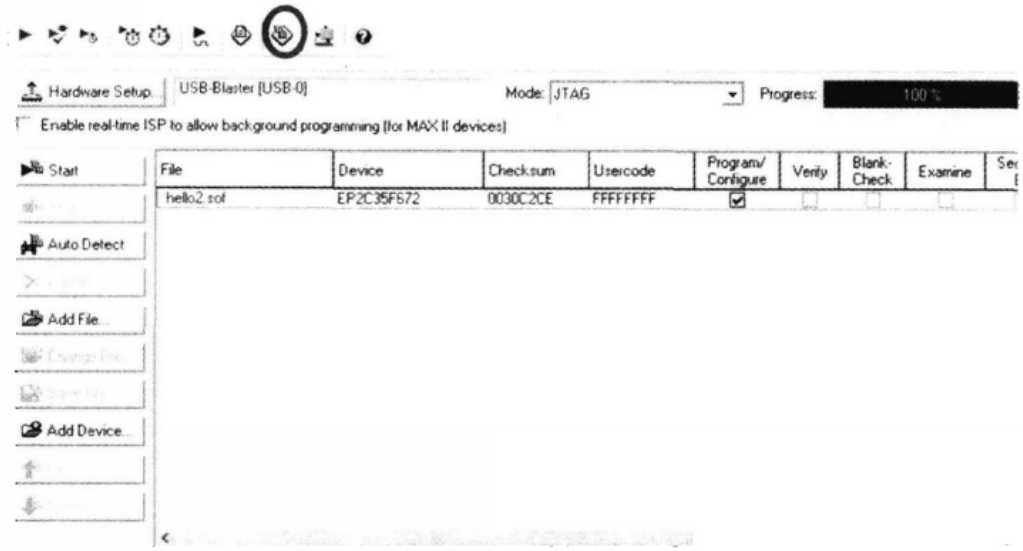


图 A-5　将编译后的位流文件下载到板上的 FPGA 芯片

(2) 体会不同的设计对 FPGA 逻辑资源率的不同要求，初步掌握如何根据 FPGA 逻辑资源的特点来优化电路的设计。

(3) 掌握管脚约束技术，熟悉电路中的输入输出和开发板上器件的连接设置方法。

(4) 掌握常用电路模块的设计方法。

扩展实验

可将 char_idx 和 seg_idx 各用一个按钮实现，每按 char_idx 一下七段管的内容改变一次，每按 seg_idx 一下选中一个七段管，被选中的七段管需要闪烁显示。此时需要考虑按下按钮时输入信号的判断，要解决如何去除干扰等问题。可阅读 DE2 开发板所提供的文档进行设计。

附录B

七段数码管计数实验

实验目的

(1) 了解用 FPGA 实现时序电路的基本原理。

(2) 了解应用 FPGA 进行开发的基本流程。

(3) 掌握常用电路如分频器、译码器和时序电路的设计与优化。

实验内容

用 DE2 开发板上的一个七段数码管实现一个计数范围为 0～9 的计数器，每隔一秒实现一次计数。顶层模块定义如图 B-1 所示。

```
module sevenseg (reset, clk, segout);
input          clk,reset;
output [6:0]     segout;
endmodule
```

图 B-1　七段数码管计数电路的顶层模块电路接口

其中各个信号定义如下。

reset：全局异步复位信号，用 DE2 上的按钮实现，当按下按钮时数码管显示为“0”。

clk：全局时钟输入信号。

segout：数码管信号输出，连接到 DE2 上的数码管，用于显示计数。

实验步骤

(1) 定义好整个电路的模块划分，包括模块的数目，各模块的接口与功能。

(2) 运行 Quartus，建立一个 Verilog/VHDL 设计项目，其中器件选择为 Altera Cyclone ® II EP2C35F672C6，如图 B-2 所示。

(3) 用硬件描述语言实现定义的各个模块，最后描述出顶层模块连接。

(4) 编写模块的仿真电路和激励，用 Modelsim 等工具进行功能仿真，若仿真不通过则寻找原因并修改相应的模块。

(5) 导入相应的管脚约束文件(.csv 文件)，确保顶层模块的信号被正确约束到相应管脚上。

(6) 单击“编译”按钮，此时 Quartus 将执行综合、映射、布局布线等步骤并最终生成 FPGA 位流文件，如图 B-3 所示。编译后的生成报告显示了模块的资源利用率、时序性能等信息。

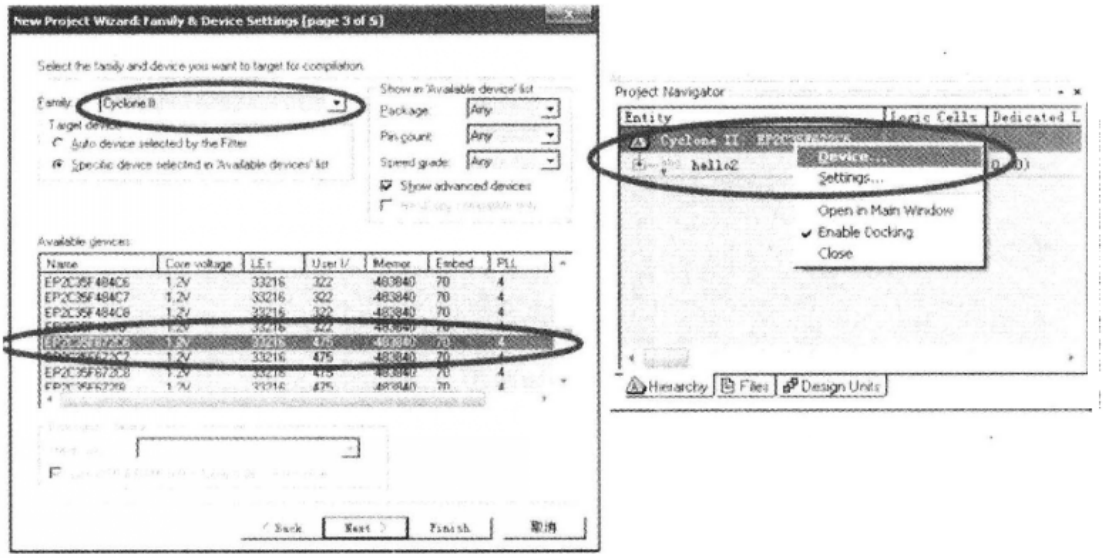


图 B-2　在 Quartus 中选择合适的 FPGA 器件

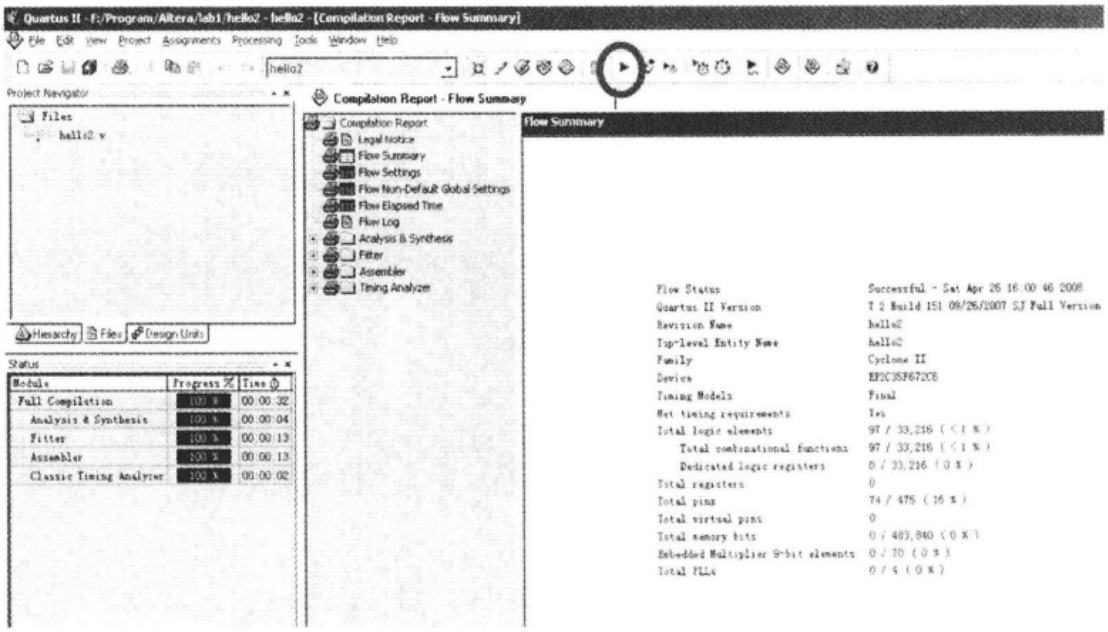


图 B-3　运行 Quartus 编译流程

(7) 启动下载程序将生成的位流文件下载到 FPGA 芯片上进行在线调试，如图 B-4 所示。若发现模块功能有误还可以利用 Quartus 的信号采样功能，例如 SignalTab 等工具进一步调试。

(8) 统计 Quartus 生成的资源利用率、时序性能和功耗分析等数据，进一步优化模块的设计。

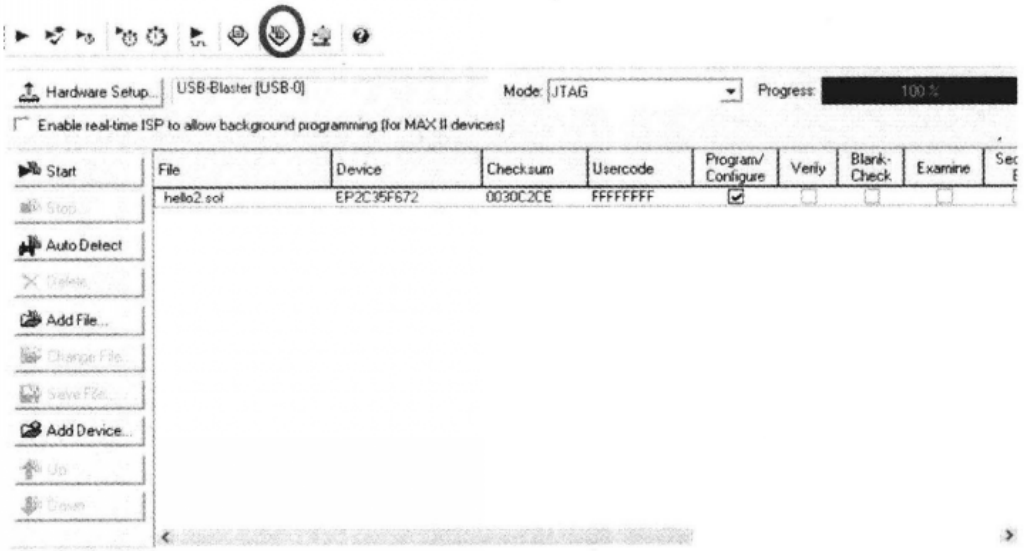


图 B-4 将编译后的位流文件下载到板上的 FPGA 芯片

实验小结

(1) 掌握时序电路的设计方法,注意根据开发板上提供的时钟和目标频率设计一个合适的时钟分频器。

(2) 了解用 FPGA 实现时序电路的设计流程,包括通过 DE2 管脚约束文件来进行管脚约束,特别注意时钟的约束。

(3) 体会不同的设计对 FPGA 逻辑资源的不同需求,初步掌握如何根据 FPGA 逻辑资源的特点来优化电路的设计。

附录C

字符串滚动显示实验

实验目的

(1) 熟悉 DE2 板子的各个部件。

(2) 熟悉 FPGA 的基本开发流程。

(3) 熟悉如何在 Quartus 中进行正确的管脚约束。

实验内容

用 DE2 板子上的 8 个七段数码管实现一个可以手动地滚动显示特定字符串("HELLO")的模块。其顶层模块定义如图 C-1 所示。

```
module scroll_display_man (SW,HEX0,HEX1,HEX2,HEX3,HEX4,HEX5,HEX6,HEX7);
input [17:0] SW;          // toggle switches
output [0:6] HEX0;        // 7 - seg displays
output [0:6] HEX1;        // 7 - seg displays
output [0:6] HEX2;        // 7 - seg displays
output [0:6] HEX3;        // 7 - seg displays
output [0:6] HEX4;        // 7 - seg displays
output [0:6] HEX5;        // 7 - seg displays
output [0:6] HEX6;        // 7 - seg displays
output [0:6] HEX7;        // 7 - seg displays
endmodule
```

图 C-1　字符串滚动显示电路的顶层模块电路接口

其中各个信号定义如下。

SW[17:0]：其中 SW[17:15]用于控制特定字符串的滚动(手动模式)，滚动方式如图 C-2 所示；SW[14:12]用于控制显示空白字符；SW[11:9]用于控制显示"H"；SW[8:6]用于控制显示"E"；SW[5:3]用于控制显示"LL"；SW[2:0]用于控制显示"O"。

HEX0～7[0:6]：数码管信号输出，连接到 DE2 上的 8 个七段数码管。

实验步骤

(1) 定义好整个电路的模块划分，包括模块的数目，各模块的接口与功能。

(2) 运行 Quartus，建立一个 Verilog/VHDL 设计项目，其中器件选择为 Altera Cyclone Ⅱ EP2C35F672C6，如图 C-3 所示。

SW_{17} SW_{16} SW_{15}	Character pattern
000	H E L L O
001	H E L L O
010	H E L L O
011	H E L L O
100	E L L O H
101	L L O H E
110	L O H E L
111	O H E L L

图 C-2 手动模式下字符串的滚动方式

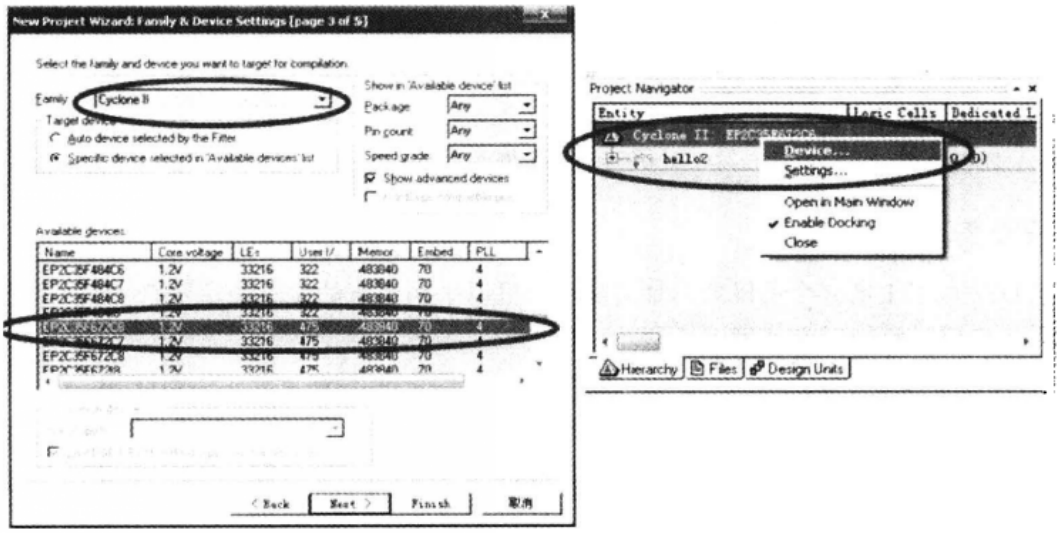


图 C-3 在 Quartus 中选择合适的 FPGA 器件

(3) 用硬件描述语言实现定义的各个模块,最后描述出顶层模块连接。

(4) 编写模块的仿真电路和激励,用 Modelsim 等工具进行功能仿真,若仿真不通过则寻找原因并修改相应的模块。

(5) 导入相应的管脚约束文件(.csv 文件),确保顶层模块的信号被正确约束到相应管脚上。

(6) 单击"编译"按钮,此时 Quartus 将执行综合、映射、布局布线等步骤并最终生成 FPGA 位流文件,如图 C-4 所示。编译后的生成报告显示了模块的资源利用率、时序性能等信息。

(7) 启动下载程序将生成的位流文件下载到 FPGA 芯片上进行在线调试,如图 C-5 所示。若发现模块功能有误还可以利用 Quartus 的信号采样功能,例如 SignalTab 等工具进一步调试。

(8) 统计 Quartus 生成的资源利用率、时序性能和功耗分析等数据,学习使用添加时序约束来进一步优化电路的性能。

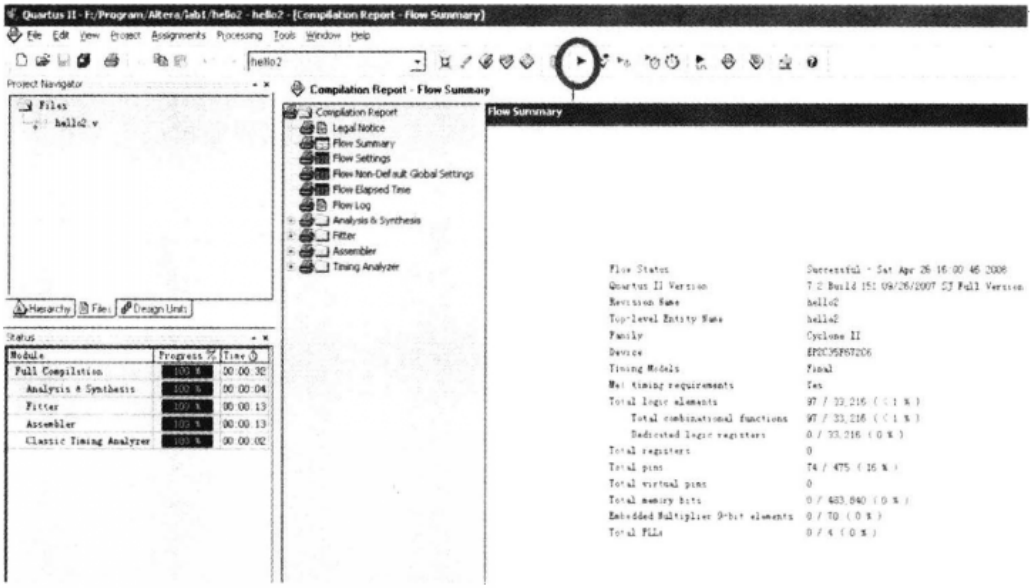


图 C-4　运行 Quartus 编译流程

图 C-5　将编译后的位流文件下载到板上的 FPGA 芯片

实验小结

(1) 熟悉用 FPGA 进行电路设计的流程和方法。

(2) 理解时序约束对电路性能的影响，通过时序约束提高电路的性能。

(3) 掌握 SignalTab 的在线调试工具。

扩展实验

用DE2板子上的8个七段数码管实现一个可以自动地滚动显示特定字符串("HELLO")的模块。每秒滚动一次,滚动方式如图C-6所示。其顶层模块定义和实现模式如图C-7所示。

```
module scroll_display_auto (CLK,SW,HEX0,HEX1,HEX2,HEX3,HEX4,HEX5,HEX6,HEX7);
input         CLK;
input [14:0] SW;          // toggle switches
output [0:6] HEX0;        // 7 - seg displays
output [0:6] HEX1;        // 7 - seg displays
output [0:6] HEX2;        // 7 - seg displays
output [0:6] HEX3;        // 7 - seg displays
output [0:6] HEX4;        // 7 - seg displays
output [0:6] HEX5;        // 7 - seg displays
output [0:6] HEX6;        // 7 - seg displays
output [0:6] HEX7;        // 7 - seg displays
endmodule
```

图C-6 扩展实验的顶层模块接口

Clock cycle	Displayed pattern
0	H E L L O
1	H E L L O
2	H E L L O
3	H E L L O
4	E L L O H
5	L L O H E
6	L O H E L
7	O H E L L
8	H E L L O
...	and so on

图C-7 自动模式下的字符串滚动方式

附录D

英文缩写对照表

ALU(arithmetic logic unit) 算术逻辑单元

AMBA(advanced microcontroller bus architecture) 先进微控制器总线架构

API(application program interface) 应用程序接口

ASCII(America Standard Code for Information Interchange) 美国信息交换标准码

ASIC(application specific integrated circuit) 专用集成电路

ASSP(application specific standard product) 标准专用产品

ATE(automatic test equipment) 自动测试设备

BGA(ball grid array) 球栅阵列封装

BLE(basic logic element) 基本逻辑单元

BSP(board support package) 板级支持包

CB(connection block) 连接盒

CLB(configurable logic block) 可配置逻辑块

CMP(chemical-mechanical polishing) 化学机械抛光技术

CPU(central processing unit) 中央处理器

CMOS(complementary metal oxide semiconductor) 互补金属氧化物半导体

CPLD(complex programmable logic device) 复杂可编程逻辑器件

DCR(device control register) 设备控制寄存器

DFM(design for manufacturability) 可制造性设计

DLL(delay locked loop) 延迟锁定环

DMA(direct memory access) 直接存储访问

DPR(dynamic partial reconfiguration) 动态部分重配置

DRAM(dynamic random access memory) 动态随机访问存储器

DRC(design rule check) 设计规则检查

EDVAC(electronic discrete variable automatic computer) 离散变量自动电子计算机

EEPROM(electrically erasable programmable read-only memory) 电擦除可编程只读存储器

ELF(executable and linkable format) 可执行可连接格式

ENIAC(electronic numerical integrator and computer) 电子数字积分计算机

EPROM(erasable programmable read-only memory) 可擦除可编程只读存储器

FIFO(first input first output) 先进先出

FIR(finite impulse response) 有限冲击响应
FPGA(field programmable gate array) 现场可编程门阵列
FP(frame pointer) 帧指针
GAL(generic array logic) 通用阵列逻辑
GCC(GNU compiler collection) GNU 套装编译器
GDS(graphical design system) 图形设计系统
GNU(GNU is not unix)
GPGPU(general purpose graphic processing unit) 通用图形处理器
GPIO(general-purpose I()) 通用输入输出
HAL(hardware abstraction layer) 硬件抽象层
HDL(hardware description language) 硬件描述语言
HSTL(high-speed transceiver logic) 高速收发逻辑
I()(input output) 输入输出
IP(intellectual property) 知识产权
JTAG(Joint Test Action Group) 联合测试行动小组
LC(logic cell) 逻辑单元
LSI(large scale integrated circuit) 大规模集成电路
LUT(look-up table) 查找表
LVCMOS(low voltage CMOS) 低电压 CMOS
LVDS(low voltage differential signal) 低电压差分信号
LVTTL(low voltage TTL) 低电压 TTL
LVS(layout versus schematics) 原理图和版图验证
MCU(micro controller unit) 微控制器
MPGA(mask programmable gate array) 掩膜可编程门阵列
MPU(micro-processor unit) 微处理器
NoC(network-on-a-chip) 片上网络
NRE(non-recurring engineering) 一次性工程费用
OCB(on-chip bus) 片上总线
OMAP(open multimedia applications platform) 开放式多媒体应用平台
OPC(optical proximity correction) 光学近似修正技术
OSI(open system interconnect) 开放式系统互联
PAL(programmable array logic) 可编程阵列逻辑
PLA(programmable logic array) 可编程逻辑阵列
PLD(programmable logic device) 可编程逻辑器件
PLL(phase locked loop) 锁相环
PSM(phase shift mask) 相移掩膜技术
PCB(printed circuit board) 印刷电路板
RTOS(real-time operating system) 实时操作系统
RTL(register transfer level) 寄存传输级

SB(switch block) 开关盒

SI(signal integrity) 信号完整性

SIMD(single instruction multiple data) 单指令多数据

SoC(system-on-a-chip) 片上系统

SoP(sum-of-product) 乘积项之和

SOPC(system on a programmable chip) 可编程片上系统

SPLD(simple programmable logic device) 简单可编程逻辑器件

SRAM(static random access memory) 静态随机访问存储器

SSI(small scale integrated circuit) 小规模集成电路

SP(stack pointer) 堆栈指针

TTL(transistor-transistor logic) 晶体管晶体管逻辑

VHDL(very-high-speed integrated circuit hardware description language) 高速集成电路硬件描述语言

VLSI(very large scale integrated circuit) 超大规模集成电路